CSC 222: Object-Oriented Programming

Spring 2012

interacting objects

- abstraction, modularization
- internal vs. external method calls
- expressions, type casting
- primitives vs. objects
- modular design: dot races
- static fields, final static fields
- private (helper) methods

1

Abstraction

abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem

- note: we utilize abstraction everyday
  do you know how a TV works? could you fix one? build one?
  do you know how an automobile works? could you fix one? build one?

abstraction allows us to function in a complex world

- we don't need to know how a TV or car works
- must understand the controls *(e.g., remote control, power button, speakers for TV)*
  (e.g., gas pedal, brakes, steering wheel for car)
- details can be abstracted away – not important for use

the same principle applies to programming

- we can take a calculation/behavior & implement as a method
  after that, don't need to know how it works – just call the method to do the job
- likewise, we can take related calculations/behaviors & encapsulate as a class

2

Abstraction examples

recall the Die class

- included the method `roll`, which returned a random roll of the Die

do you remember the formula for selecting a random number from the right range?

WHO CARES?!? Somebody figured it out once, why worry about it again?

SequenceGenerator class

- included the method `randomSequence`, which returned a random string of letters

you don't know enough to code it, but you could use it!

Circle, Square, Triangle classes

- included methods for drawing, moving, and resizing shapes

again, you don't know enough to code it, but you could use it!

Modularization

modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways

- early computers were hard to build – started with lots of simple components (e.g., vacuum tubes or transistors) and wired them together to perform complex tasks
- today, building a computer is relatively easy – start with high-level modules (e.g., CPU chip, RAM chips, hard drive) and plug them together

the same advantages apply to programs

- if you design and implement a method to perform a well-defined task, can call it over and over within the class
- likewise, if you design and implement a class to model a real-world object's behavior, then you can reuse it whenever that behavior is needed (e.g., Die for random values)

Code reuse can occur within a class

one method can call another method

- a method call consists of method name + any parameter values in parentheses (as shown in BlueJ when you right-click and select a method to call)

```
this.methodName(paramValue1, paramValue2, …);
```

- calling a method causes control to shift to that method, executing its code
- if the method returns a value (i.e., a return statement is encountered), then that return value is substituted for the method call where it appears

```
public class Die {
 . . .

  public int getNumberOfSides() {
      return this.numSides;
  }

  public int roll() {
      this.numRolls++;
      return (int)(Math.random()*this.getNumberOfSides() + 1);
  }
}
```

here, the number returned by the call to `getNumberOfSides` is used to generate the random roll

5

ESPTester class

```
public class ESPTester {
  ...

  public double percentageCorrect() {
    // CALCULATE & RETURN %
  }

  public String ESPverdict() {
    double percent = this.percentageCorrect();
    double expected = 100.0/this.maxNumber;

    if (percent > expected) {
      return "You just might have ESP!";
    }
    else {
      return "You definitely do not have ESP.";
    }
  }
}
```

when a method calls another method from within the same object, we call that an *internal method call*

for HW1:

- you wrote `percentageCorrect`, which calculated & returned the % of correct guesses
- `ESPVerdict` makes an internal method call to call to this method to get & compare the percentage
- note: we could have declared the `percentageCorrect` method to be private – it would still have been accessible to other methods, but not to the outside user

6

Expressions and types

in general, Java operations are *type-preserving*

- if you add two `int`s, you get an `int` — `2 + 3 → 5`
- if you add two `double`s, you get a `double` — `2.5 + 3.5 → 6.0`
- if you add two `String`s, you get a `String` — `"foo" + "2u" → "foo2u"`

this applies to division as well

- if you divide two `int`s, you get an `int` – any fractional part is discarded!

```
8/4 → 2          9/4 → (2.25) → 2        -9/4 → (-2.25) → -2
1/2 → ???        999/1000 → ???          x/(x+1) → ???
```

for mixed expressions, the more specific value is converted first

```
3.2 + 1 → (3.2 + 1.0) → 4.2

"x = " + 5 → "x = " + "5" → "x = 5"
```

FYI: the % operator gives the remainder after performing `int` division

12 % 2 → 0   13 % 2 → 1   18 % 5 → ???

7

int vs. real division

must be very careful when dividing `int` values

```
double percent = this.numCorrect/this.numGuesses;
```

Solution 1: declare the fields to be `double` instead of `int`

KLUDGY! if a value is an integer, declare it to be an int!

Solution 2: introduce a real value into the equation

```
double percent = 1.0*this.numCorrect/this.numGuesses;
```

SLIGHTLY LESS KLUDGY! raises questions in the reader's mind

Solution 3: cast (convert) one of the `int` values into a `double`   `(NEWTYPE)VALUE`

```
double percent = (double)this.numCorrect/this.numGuesses;
```

GOOD! makes the type conversion clear to reader

8

Primitive types vs. object types

primitive types are predefined in Java, e.g., `int`, `double`, `boolean`

object types are those defined by classes, e.g., `Circle`, `Die`

- so far, our classes have utilized primitives for fields/parameters/local variables
- as we define classes that encapsulate useful behaviors, we will want build on them

when you declare a variable of primitive type, memory is allocated for it

- to store a value, simply assign that value to the variable

```
int x;                          double height = 72.5;
x = 0;
```

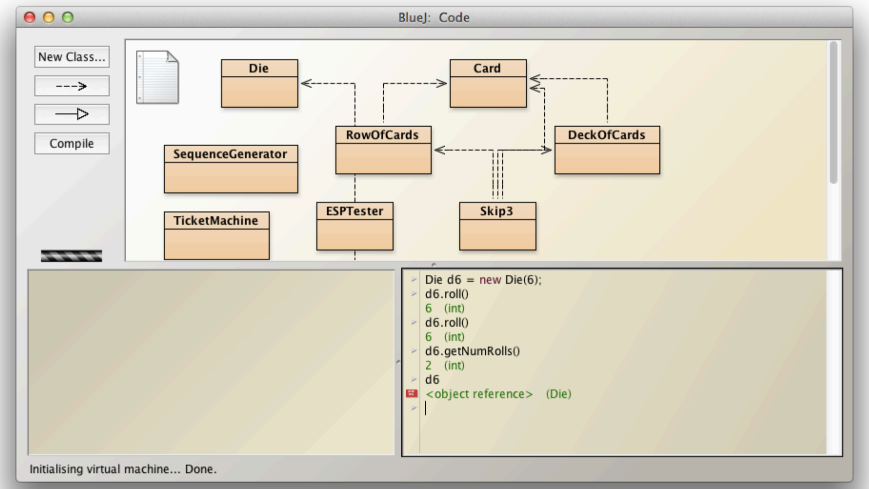
when you declare a variable of object type, it is NOT automatically created

- to initialize, must call its constructor: `OBJECT = new CLASS(`*`PARAMETERS`*`);`
- to call a method: `OBJECT.METHOD(`*`PARAMETERS`*`)`

```
Circle circle1;                 Die d8 = new Die(8);
circle1 = new Circle();         System.out.println( d8.roll() );
circle1.changeColor("red");
```

BlueJ's code pad

the code pad allows you to enter Java statements to create objects and call methods on those objects



```
Die d6 = new Die(6);
d6.roll()
6   (int)
d6.roll()
6   (int)
d6.getNumRolls()
2   (int)
d6
<object reference>   (Die)
```

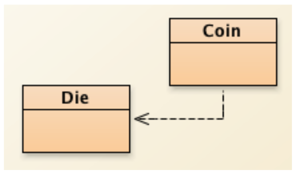
Initialising virtual machine... Done.

Coin class

we can define classes with fields that are objects of other classes

- `Coin` class has a 2-sided `Die` as a field
- `Coin` constructor must construct the `Die` object to initialize that field
- `flip` method makes an *external method call* on the `Die` object

suppose we wanted to add a getNumFlips method?



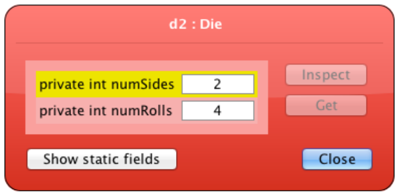
```
public class Coin {
  private Die d2;

  public Coin() {
    this.d2 = new Die(2);
  }

  public String flip() {
    int rollResult = this.d2.roll();
    if (rollResult == 1) {
      return "HEADS";
    }
    else {
      return "TAILS";
    }
  }
}
```

11

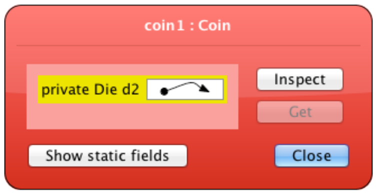
primitive types vs. object types

internally, primitive and reference types are stored differently

- when you inspect an object, any primitive fields are shown as boxes with values
- when you inspect an object, any object fields are shown as pointers to other objects



private int numSides 2
private int numRolls 4



private Die d2

- of course, you can further inspect the contents of object fields

we will consider the implications of primitives vs. objects later

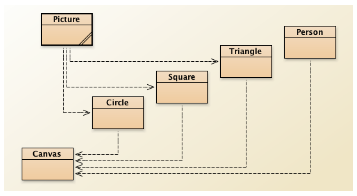
12

House example

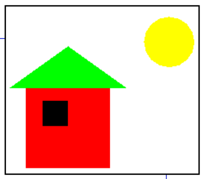
recall the picture of a house

- involved creating 4 shapes & adjusting their colors, sizes & positions
- instead of doing this manually through BlueJ, we could define a class that automates these steps

- the `Picture` class has fields for each of the shapes in the picture
- in the `draw` method, each shape is created (by calling its constructor) and adjusted (by calling methods)



```
public class Picture {
  private Square wall;
  private Square window;
  private Triangle roof;
  private Circle sun;

  public Picture() {
    this.wall = new Square();
    this.window = new Square();
    this.roof = new Triangle();
    this.sun = new Circle();
  }

  public void draw() {
    this.wall.moveVertical(80);
    this.wall.changeSize(100);
    this.wall.makeVisible();

    this.window.changeColor("black");
    this.window.moveHorizontal(20);
    this.window.moveVertical(100);
    this.window.makeVisible();

    this.roof.changeSize(50, 140);
    this.roof.moveHorizontal(60);
    this.roof.moveVertical(70);
    this.roof.makeVisible();

    this.sun.changeColor("yellow");
    this.sun.moveHorizontal(180);
    this.sun.moveVertical(-10);
    this.sun.changeSize(60);
    this.sun.makeVisible();
  }
}
```



13

Dot races

consider the task of simulating a dot race (as on stadium scoreboards)

- different colored dots race to a finish line
- at every step, each dot moves a random distance
- the dot that reaches the finish line first wins!

behaviors?

- create a race (dots start at the beginning)
- step each dot forward a random amount
- access the positions of each dot
- display the status of the race
- reset the race

we could try modeling a race by implementing a class directly

- store positions of the dots in fields
- have each method access/update the dot positions

BUT: lots of details to keep track of; not easy to generalize

14

A modular design

instead, we can encapsulate all of the behavior of a dot in a class

`Dot` class:	create a `Dot` (with a given color, maximum step size)
	access the dot's position
	take a step
	reset the dot back to the beginning
	display the dot's color & position

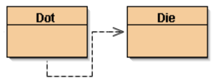
once the `Dot` class is defined, a `DotRace` will be much simpler

`DotRace` class:	create a `DotRace` (with same maximum step size for both dots)
	access either dot's position
	move both dots a single step
	reset both dots back to the beginning
	display both dots' color & position

Dot class

more naturally:

- fields store a Die (for generating random steps), color & position



- constructor creates the Die object and initializes the color and position fields

- methods access and update these fields to maintain the dot's state

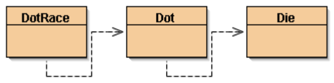
CREATE AND PLAY

```
public class Dot {
    private Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color, int maxStep) {
        this.die = new Die(maxStep);
        this.dotColor = color;
        this.dotPosition= 0;
    }

    public void step() {
        this.dotPosition += this.die.roll();
    }

    public void reset() {
        this.dotPosition = 0;
    }

    public int getPosition() {
        return this.dotPosition;
    }

    public void showPosition() {
        System.out.println(this.dotColor + ": " +
                           this.dotPosition);
    }
}
```

DotRace class

using the `Dot` class, a `DotRace` class is straightforward

- fields store the two Dots



- constructor creates the Dot objects, initializing their colors and max steps

- methods utilize the Dot methods to produce the race behaviors

CREATE AND PLAY

ADD ANOTHER DOT?

```
public class DotRace {
  private Dot redDot;
  private Dot blueDot;

  public DotRace(int maxStep) {
    this.redDot = new Dot("red", maxStep);
    this.blueDot = new Dot("blue", maxStep);
  }

  public int getRedPosition() {
    return this.redDot.getPosition();
  }

  public int getBluePosition() {
    return this.blueDot.getPosition();
  }

  public void step() {
    this.redDot.step();
    this.blueDot.step();
  }

  public void showStatus() {
    this.redDot.showPosition();
    this.blueDot.showPosition();
  }

  public void reset() {
    this.redDot.reset();
    this.blueDot.reset();
  }
}
```

17

Adding a finish line

suppose we wanted to place a finish line on the race

- what changes would we need?

could add a field to store the goal distance

- user specifies goal distance along with max step size in constructor call
- step method would not move if either dot has crossed the finish line

```
public class DotRace {
    private Dot redDot;
    private Dot blueDot;
    private int goalDistance;   // distance to the finish line

    public DotRace(int maxStep, int goal) {
        this.redDot = new Dot("red", maxStep);
        this.blueDot = new Dot("blue", maxStep);
        this.goalDistance = goal;
    }

    public int getGoalDistance() {
        return this.goalDistance;
    }

    . . .
```

18

Adding a finish line

`step` method needs a 3-way conditional:

- either blue crossed or red crossed or the race is still going on

```
public void step() {
    if (this.getBluePosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else if (this.getRedPosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        this.redDot.step();
        this.blueDot.step();
    }
}
```

ugly! we want to avoid duplicate code

fortunately, Java provides *logical operators* for just such cases

`(TEST1 || TEST2)` evaluates to true if either `TEST1` **OR** `TEST2` is true

`(TEST1 && TEST2)` evaluates to true if either `TEST1` **AND** `TEST2` is true

`(!TEST)` evaluates to true if `TEST` is **NOT** true

19

Adding a finish line

here, could use `||` to avoid duplication

- print message if *either* blue *or* red has crossed the finish line

```
public void step() {
    if (this.getBluePosition() >= this.goalDistance ||
          this.getRedPosition() >= this.goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        this.redDot.step();
        this.blueDot.step();
    }
}
```

warning: the tests that appear on both sides of `||` and `&&` must be complete Boolean expressions

`(x == 2 || x == 12)` OK

`(x == 2 || 12)` BAD!

note: we could have easily written `step` using `&&`

- move dots if *both* blue *and* red dots have failed to cross finish line

```
public void step() {
    if (this.getBluePosition() < this.goalDistance &&
          this.getRedPosition() < this.goalDistance) {
        this.redDot.step();
        this.blueDot.step();
    }
    else {
        System.out.println("The race is over!");
    }
}
```

20

Further changes

EXERCISE: make these modifications to your `DotRace` class

- add the `goalDistance` field
- modify the constructor to include the goal distance
- add an accessor method for viewing the goal distance
- add an if statement to the step method to recognize the end of the race

what if we wanted to display the dot race visually?

- could utilize the `Circle` class to draw the dots
- unfortunately, `Circle` only has methods for relative movements

  we need a `Circle` method for absolute movement (based on a Dot's position)

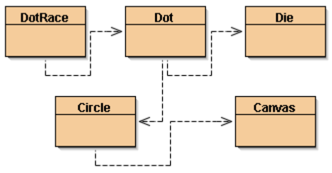
```
/**
 * Move the circle to a specific location on the canvas.
 *   @param xpos the new x-coordinate for the circle
 *   @param ypos the new y-coordinate for the circle
 */
public void moveTo(int xpos, int ypos) {
    this.erase();
    this.xPosition = xpos;
    this.yPosition = ypos;
    this.draw();
}
```

21

Adding graphics

due to our modular design, changing the display is easy

- each Dot object will maintains and display its own `Circle` image



- add `Circle` field
- constructor creates the `Circle` and sets its color
- `showPosition` moves the `Circle` (instead of displaying text)

```
public class Dot {
  private Die die;
  private String dotColor;
  private int dotPosition;
  private Circle dotImage;

  public Dot(String color, int maxStep) {
     this.die = new Die(maxStep);
    this.dotColor = color;
    this.dotPosition= 0;
     this.dotImage = new Circle();
     this.dotImage.changeColor(color);
  }

  public void step() {
    this.dotPosition += this.die.roll();
  }

  public void reset() {
    this.dotPosition = 0;
  }

  public int getPosition() {
    return this.dotPosition;
  }

  public void showPosition() {
    this.dotImage.moveTo(this.dotPosition, 68);
    this.dotImage.makeVisible();
  }
}
```

22

Graphical display

note: no modifications are necessary in the `DotRace` class!!!

- this shows the benefit of modularity
- not only is modular code easier to write, it is easier to change/maintain
- can isolate changes/updates to the class/object in question
- to any other interacting classes, the methods look the same

EXERCISE: make these modifications to the `Dot` class

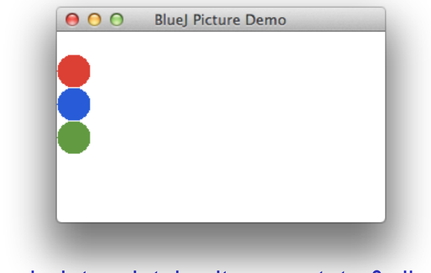
- add `Circle` field
- create and change color in constructor
- modify `showPosition` to move and display the circle

23

Better graphics

the graphical display is better than text, but still primitive

- dots are drawn on top of each other (same yPositions)
- would be nicer to have the dots aligned vertically (different yPositions)



PROBLEM: each dot maintains its own state & displays itself

- thus, each dot will need to know what yPosition it should have
- but yPosition depends on what order the dots are created in DotRace
  (e.g., 1st dot has yPosition = 68, 2nd dot has yPosition = 136, …)
- *how do we create dots with different yPositions?*

24

Option 1: Dot parameters

we could alter the Dot class constructor

- takes an additional int that specifies the dot number
- the dotNumber can be used to determine a unique yPosition
- in DotRace, must pass in the number when creating each of the dots

```
public class Dot {
  private Die die;
  private String dotColor;
  private int dotPosition;
  private Circle dotImage;
  private int dotNumber;

  public Dot(String color, int maxStep, int num) {
    this.die = new Die(maxStep);
    this.dotColor = color;
    this.dotPosition= 0;
    this.dotImage = new Circle();
    this.dotImage.changeColor(color);
    this.dotNumber = num;
  }

  . . .

  public void showPosition() {
    this.dotImage.moveTo(this.dotPosition,
                         68*this.dotNumber);
    this.dotImage.makeVisible();
  }
}
```

```
public class DotRace {
  private Dot redDot;
  private Dot blueDot;
  private Dot greenDot;

  public DotRace(int maxStep) {
    this.redDot = new Dot("red", maxStep, 1);
    this.blueDot = new Dot("blue", maxStep, 2);
    this.greenDot = new Dot("green", maxStep, 3);
  }

  . . .
}
```

this works, but is inelegant

- why should DotRace have to worry about dot numbers?
- the Dot class should be responsible

Option 2: a static field

better solution: have each dot keep track of its own number

- this requires a new dot to know how many dots have already been created
- this can be accomplished in Java via a *static field*

```
private static TYPE FIELD = VALUE;
```

- such a declaration creates and initializes a field that is shared by all objects of the class
- when the first object of that class is created, the field is initialized via the assignment
- subsequent objects simply access the existing field

```
public class Dot {
  private Die die;
  private String dotColor;
  private int dotPosition;
  private Circle dotImage;

  private static int nextAvailable = 1;
  private int dotNumber;

  public Dot(String color, int maxStep) {
    this.die = new Die(maxStep);
    this.dotColor = color;
    this.dotPosition= 0;
    this.dotImage = new Circle();
    this.dotImage.changeColor(color);

    this.dotNumber = Dot.nextAvailable;
    Dot.nextAvailable++;
  }

  . . .

  public void showPosition() {
    this.dotImage.moveTo(this.dotPosition,
                         68*this.dotNumber);
    this.dotImage.makeVisible();
  }
}
```

since static fields belong to the class, access as `CLASS.FIELD`

try it!	how do static fields appear when you inspect?	could `die` be static?

Magic numbers

`showPosition` shifts each new dot down 68 pixels in the display

- why 68?

having a "magic number" appear in the code without apparent reason is bad

- unclear to the reader
- difficult to maintain

ideally, could use an accessor method on the `dotImage` to get its diameter

```
public class Dot {
  private Die die;
  private String dotColor;
  private int dotPosition;
  private Circle dotImage;

  private static int nextAvailable = 1;
  private int dotNumber;

  public Dot(String color, int maxStep) {
    this.die = new Die(maxStep);
    this.dotColor = color;
    this.dotPosition= 0;
    this.dotImage = new Circle();
    this.dotImage.changeColor(color);

    this.dotNumber = Dot.nextAvailable;
    Dot.nextAvailable++;
  }

  . . .

  public void showPosition() {
    this.dotImage.moveTo(this.dotPosition,
                         68*this.dotNumber);
    this.dotImage.makeVisible();
  }
}
```

```
dotImage.moveTo(dotPosition, dotImage.getDiameter()*dotNumber);
```

but the Circle class does not provide accessor methods! ☹

27

Final static

if can't get the diameter from Circle, will need to store in Dot

- store the diameter in a field, can then use to set the circle size (in the constructor) and space dots (in `showPosition`)
- since all dots will have the same size, make the field *static*
- since the value should not change, can also make it *final*
- once initialized, a final value cannot be changed (any attempt causes a compiler error)

```
public class Dot {
  private Die die;
  private String dotColor;
  private int dotPosition;
  private Circle dotImage;

  private static int nextAvailable = 1;
  private int dotNumber;

  private static int dotSize = 60;

  public Dot(String color, int maxStep) {
    this.die = new Die(maxStep);
    this.dotColor = color;
    this.dotPosition= 0;
    this.dotImage = new Circle();
    this.dotImage.changeColor(color);
    this.changeSize(Dot.dotSize);

    this.dotNumber = Dot.nextAvailable;
    Dot.nextAvailable++;
  }

  . . .

  public void showPosition() {
    this.dotImage.moveTo(this.dotPosition,
                         Dot.dotSize*this.dotNumber);
    this.dotImage.makeVisible();
  }
}
```

28

Conditional repetition

running a dot race is a tedious tasks

- you must call `step` and `showStatus` repeatedly to see each step in the race

a better solution would be to automate the repetition

in Java, a while loop provides for *conditional repetition*

- similar to an if statement, behavior is controlled by a condition (Boolean test)
- as long as the condition is true, the code in the loop is executed over and over

```
while (BOOLEAN_TEST) {
    STATEMENTS TO BE EXECUTED
}
```

```
int num = 1;
while (num < 5) {
    System.out.println(num);
    num++;
}
```

```
int x = 10;
int sum = 0;
while (x > 0) {
    sum += x;
    x -= 2;
}
System.out.println(sum);
```

29

`runRace` method

can define a `DotRace` method with a while loop to run the entire race

in pseudocode:

```
RESET THE DOT POSITIONS
SHOW THE DOTS
while (NO DOT HAS WON) {
   HAVE EACH DOT TAKE A STEP
   SHOW THE DOTS
}
```

```
/**
 * Conducts an entire dot race, showing the status
 * after each step.
 */
public void runRace() {
  this.reset();
  this.showStatus();
  while (this.getRedPosition() < this.goalDistance &&
         this.getBluePosition() < this.goalDistance) {
     this.step();
     this.showStatus();
  }
}
```

30

Private methods

once we have `runRace`, we might want to limit low-level interaction

declaring a method to be `private` makes it:

- invisible to the user (e.g., doesn't show up in BlueJ)
- but still usable by other methods (we can refer to it as a *helper method*)

```
public class DotRace {
  private Dot redDot;
  private Dot blueDot;
  private int goalDistance;

  public DotRace(int maxStep, int goal) {
    this.redDot = new Dot("red", maxStep);
    this.blueDot = new Dot("blue", maxStep);
    this.goalDistance = goal;
  }

  public int getGoalDistance() {
    return this.goalDistance;
  }

  public void runRace() {
    this.reset();
    this.showStatus();
    while (this.getRedPosition() < this.goalDistance &&
           this.getBluePosition() < this.goalDistance) {
      this.step();
      this.showStatus();
    }
  }

  //////////////////////////

  private int getRedPosition() { … }

  private int getBluePosition() { … }

  private void step() { … }

  private void showStatus() { … }

  private void reset() { … }
}
```

31

Class/object summary

a class defines the content and behavior of a new type

- *fields*: variables that maintain the state of an object of that class
  - fields persist as long as the object exists, accessible to all methods
  - if want a single field to be shared by all objects in a class, declare it to be *static*
  - if it is a constant value that won't change, declare it to be *final static*

  to store a primitive value: declare a variable and assign it a value
  to store an object: declare a variable, call a constructor and assign to the variable

- *methods*: collections of statements that implement behaviors
  - methods will usually access and/or update the fields to produce behaviors
  - statements: assignment, println, return, if, if-else, while, method call (internal & external)

  parameters are variables that store values passed to a method (allow for generality)
  – parameters persist only while the method executes, accessible only to the method
  local variables are variables that store temporary values within a method
  – local variables exist from point they are declared to the end of method execution
  private methods are invisible to the user, but can be called by other methods in the class

- *constructors:* methods (same name as class) that create and initialize an object
  - a constructor assigns initial values to the fields of the object (can have more than one)

32