Day 01: Introduction to OOP

Slide Credits: *Internet / Chetan Arora*

Backpack

Website: https://classroom.google.com/u/1/c/MTI2MTkzMTE1Njk1

- **Class code: mge6grc**

- Assignment submission
- Course related discussion

Methodology

- Module Duration: 17th , 18th and 19st August (3 days module)

- Lectures:

  CSE & CSD:
  Lectures: Monday, Tuesday, Wednesday (11:30am-1:00 pm)
  Lab: Take Home

  ECE, CSSS & CSD, CSAM, CSAI, rest others:
  Lectures: Monday, Tuesday, Wednesday (11:30am-1:00 pm)
  Labs: Take Home

  - Lab assignment will be declared after 2 pm.
  - 2 practice Labs, 1 evaluated Lab and 1 quiz

Evaluation

- Homework
  - 1 Lab. Assignment to be released in the third Lab. To be submitted on the Backpack/Classroom

- Quiz on the last day

- These evaluated assignment and quiz carries a weightage in the upcoming Advanced Programming Course this Monsoon.

Rules

- No deadline extensions
- No re-exam.
- Cheating or copying in a exam: Minimum: zero in the exam, Maximum: depends upon the extent of offense.
- Cheating/copying/plagiarism in assignment: Minimum: grade reduction, Maximum: depends upon the extent of offense.
- Assignment have to be submitted – even if you delayed it! Otherwise written exam will not be graded (i.e., will earn 0 points)

Evaluation for Assignment and Homework

- Group of TAs: on the course website
- TA evaluates assignments and exam.
- Any question related to evaluation should be taken to TA first. Escalate to Teaching Fellow if required. Escalate to instructor only after the first two options exhausted.
- Questions related to exam also goes to evaluating TA first.
- Any email to the instructor must have subject text start with [AP Refresher]. Mail will be deleted without reading if protocol not followed. Send reminder after 24 hours if unanswered.

Module Objectives

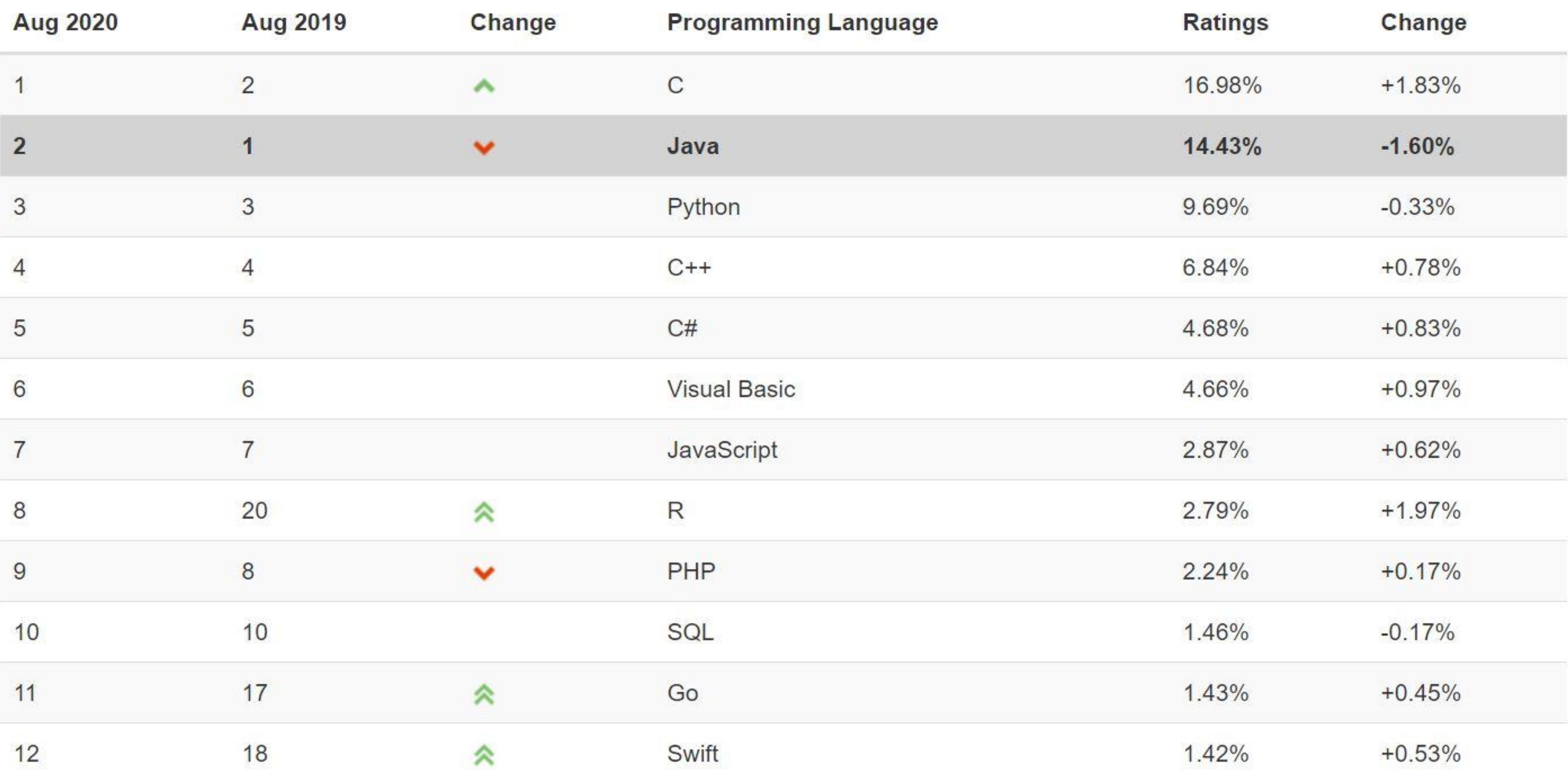
- Prerequisite:
  - Familiarity with basic programming in Java.

- Topics to be covered in this 3 days module
  1. Introduction to Object Oriented Programming (OOP) in Java
  2. What are classes and objects
  3. Using classes and objects
  4. Class methods and fields
  5. GUI programming

Textbooks and References

- Textbooks
  - Core Java - Volumes I and II. by Horstmann and Cornell.
- Reference Books:
  - Programming Pearls and More programming pearls - confessions of a coder. by Jon Louis Bentley.
  - Program Development in Java - Abstraction, Specification, and Object-Oriented Design. by Liskov and Guttag.

Popularity of Java



Aug 2020	Aug 2019	Change	Programming Language	Ratings	Change
1	2	⌃	C	16.98%	+1.83%
2	**1**	⌄	**Java**	**14.43%**	**-1.60%**
3	3		Python	9.69%	-0.33%
4	4		C++	6.84%	+0.78%
5	5		C#	4.68%	+0.83%
6	6		Visual Basic	4.66%	+0.97%
7	7		JavaScript	2.87%	+0.62%
8	20	⇈	R	2.79%	+1.97%
9	8	⌄	PHP	2.24%	+0.17%
10	10		SQL	1.46%	-0.17%
11	17	⇈	Go	1.43%	+0.45%
12	18	⇈	Swift	1.42%	+0.53%

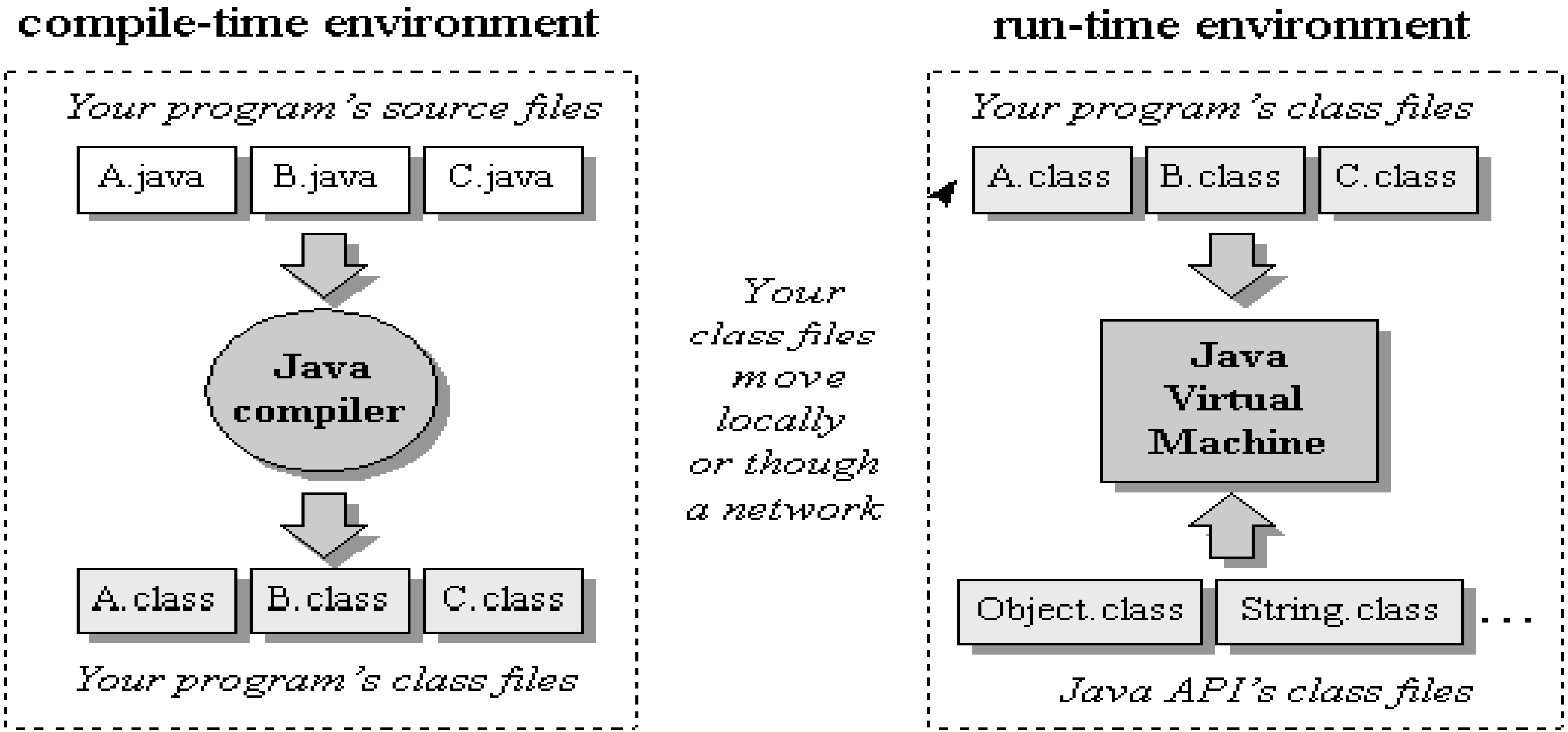
TIOBE Index for August 2020

11 Buzzwords for Java

- Simple
- Object Oriented
- Network Savvy
- Robust
- Secure
- Architecture Neutral
- Portable
- Interpreted
- High performance
- Multithreaded
- Dynamic

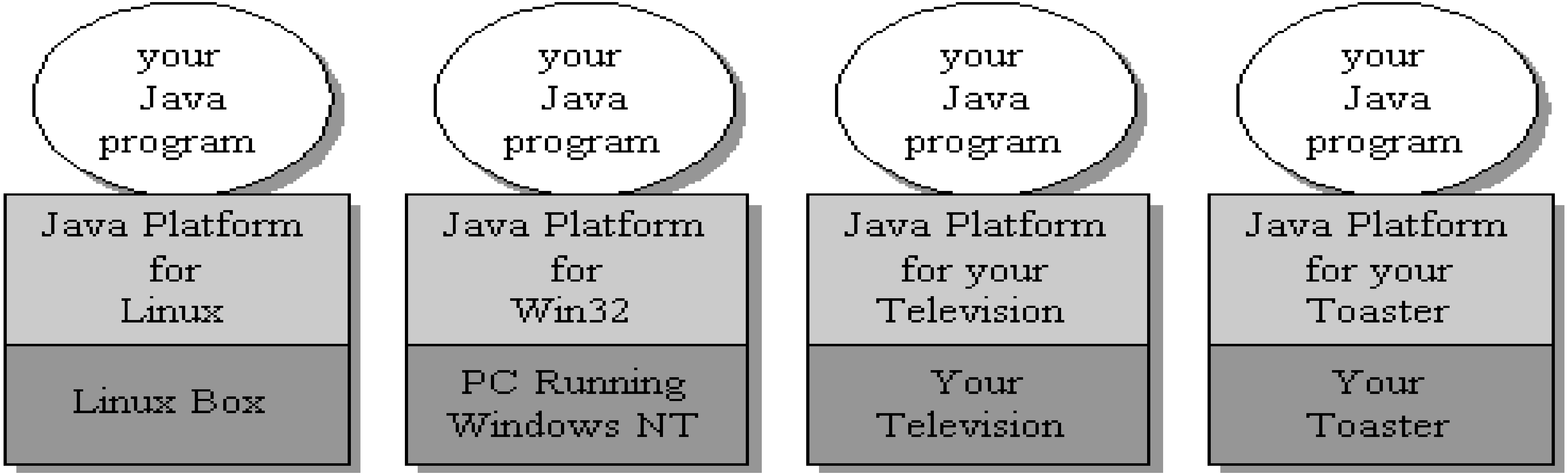
Java Architecture

- Java programming environment



Java Architecture

- Java platform (Java Virtual Machine + Java API)



Structured Programming

- Methods define the structure of the programs, they are basic building blocks

- Data has secondary role, it is just something that is passed around.

Object Oriented Programming

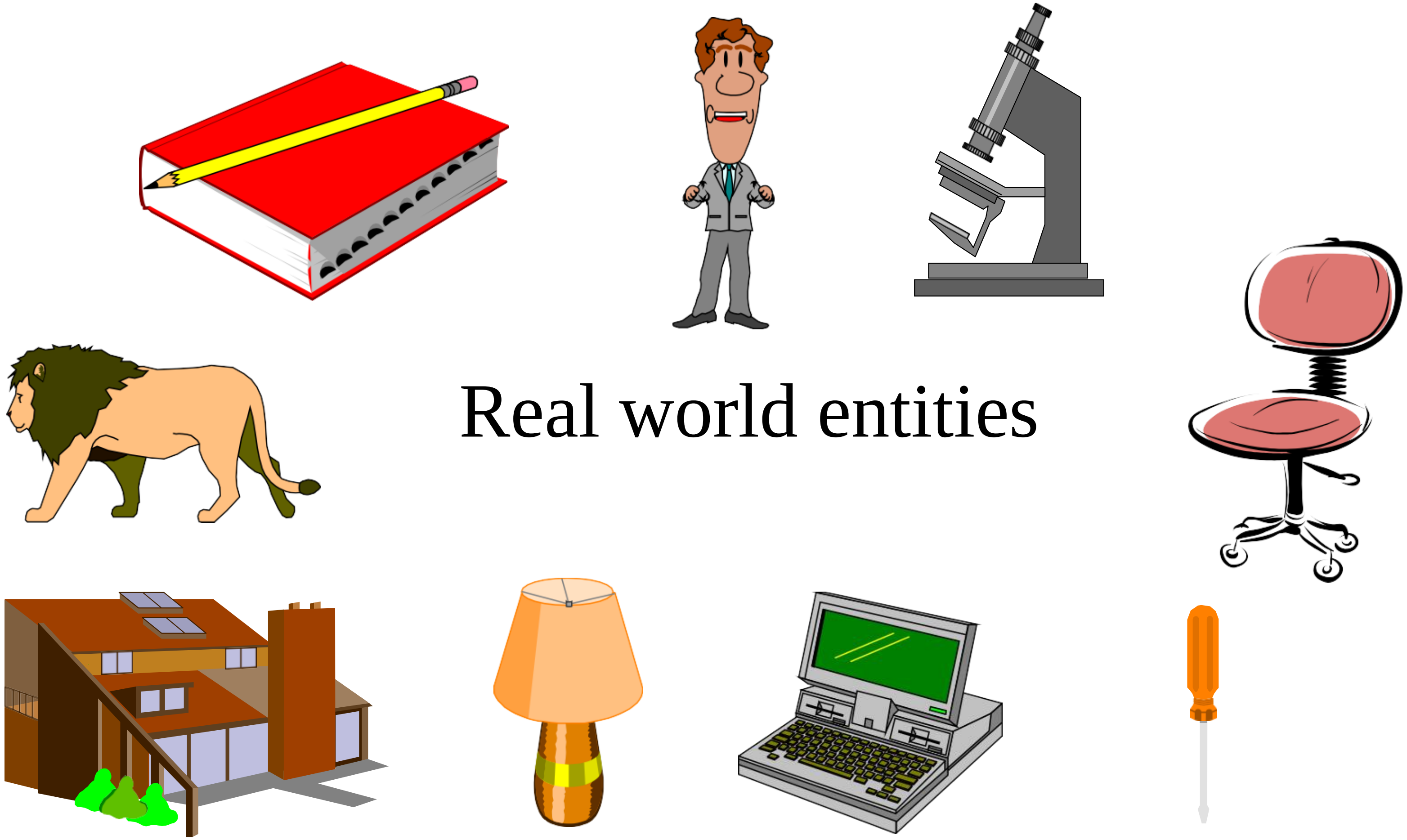
- The data has the principal role

- Methods belong to the data, without the data, the method does not have any meaning (Except static methods)

- Data and methods together make up the object.

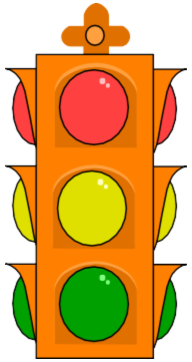
- OOP tries to model the real world.

Real World

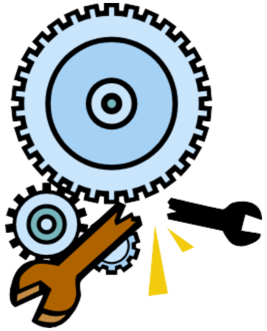


Real world entities

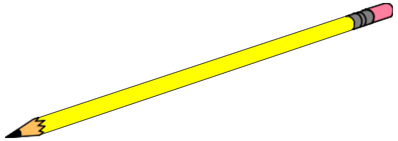
Objects have states



Red

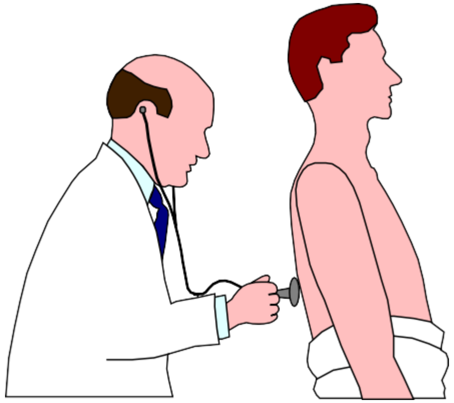


Broken

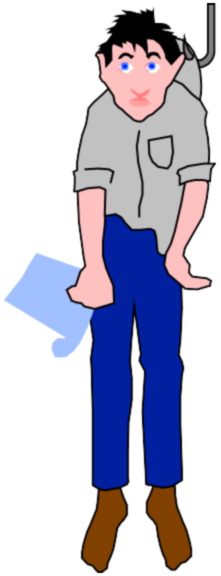


Lying

Happy

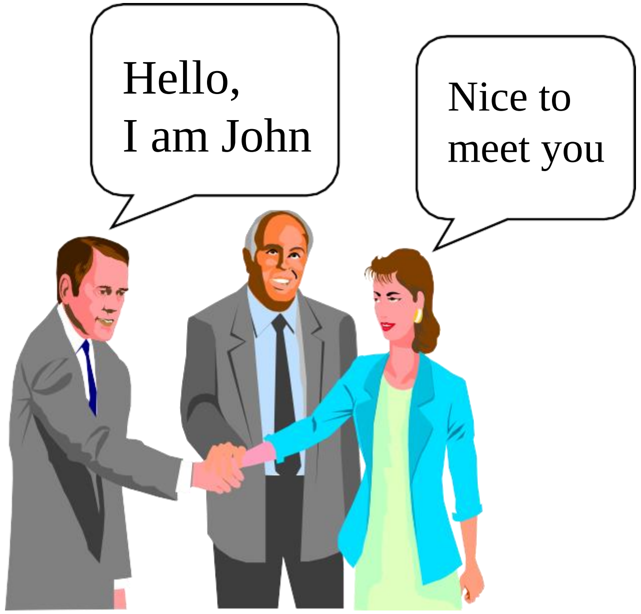


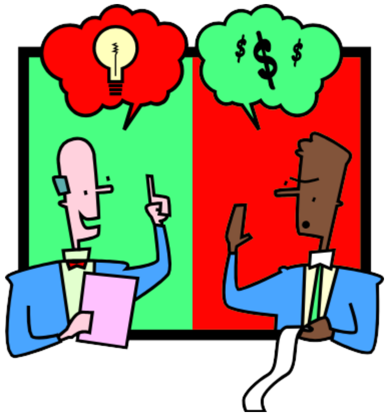
ill

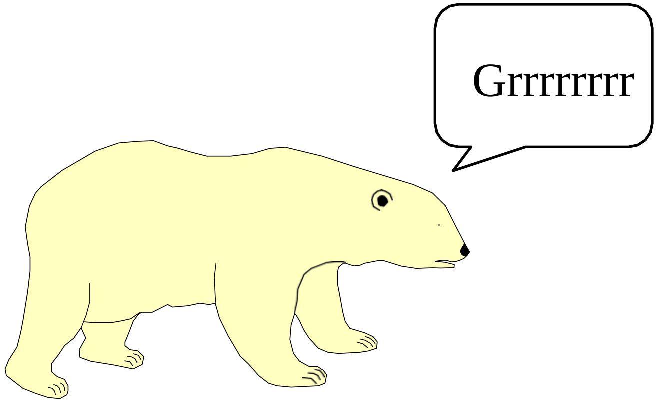


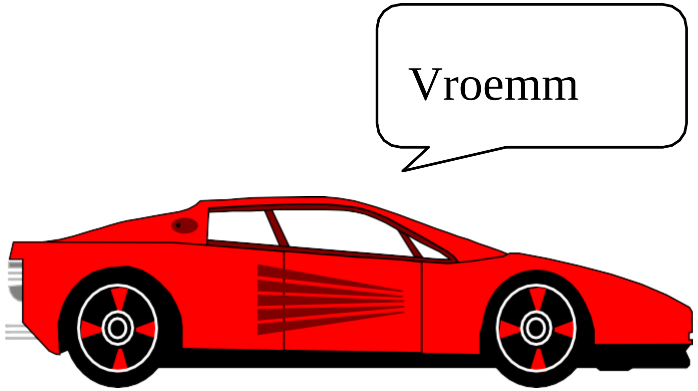
Hooked

Objects have behavior









Object Properties

- Identity
- State
- Behavior

on

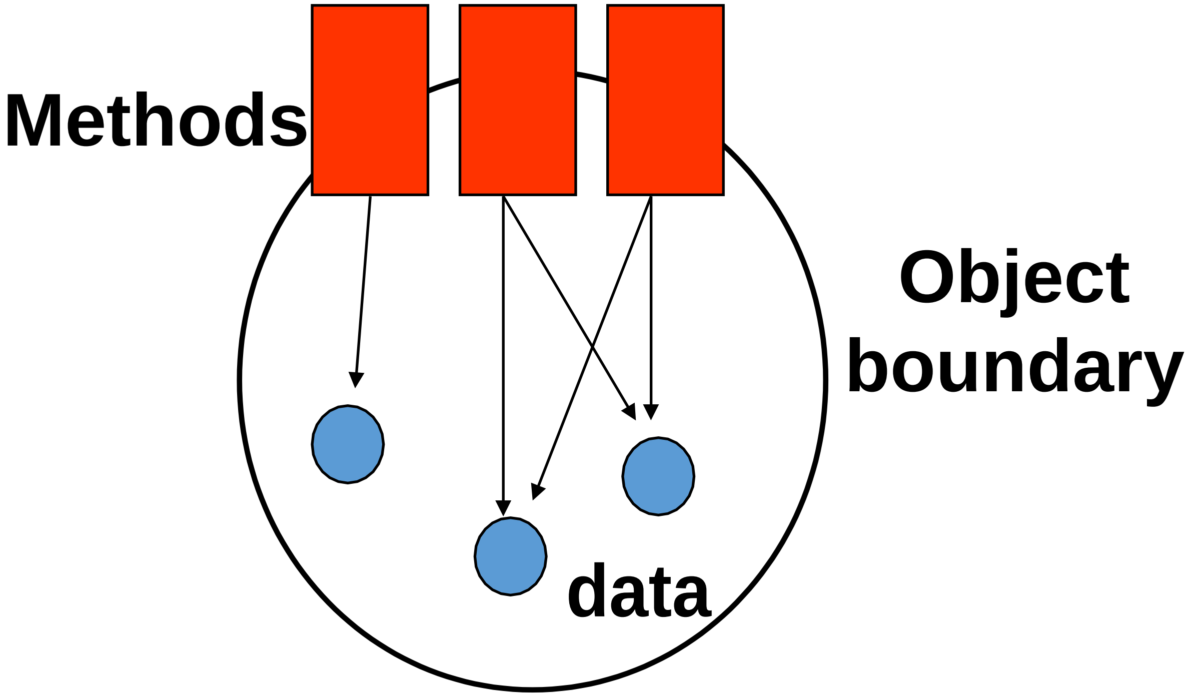
off

myLamp

Object is an **abstraction** of a real world entity

OOP Modelling: Objects and Classes

- Each object represents an **abstraction**
  - A “black box”: hides details we do not care about
  - Allows a programmer to control programs’ complexity - only think about salient features



OOP Modelling: Objects and Classes

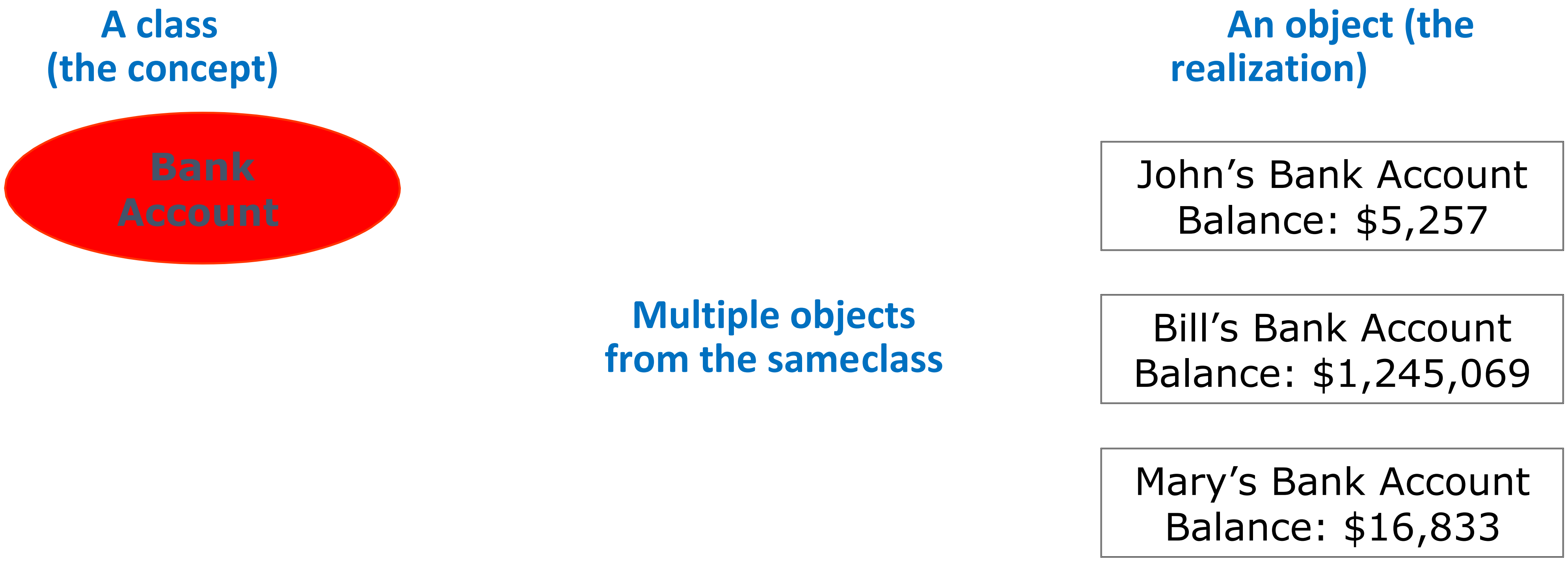
- Class – Category/Bluprint/Contract
  - Properties/states
  - Functionality/Services (examines/alters state)

data

methods

- Object - Individual/unique thing (an instance of a class)
  - Particular value for each property/state
  - Functionality of all members of class

OOP Modelling: Objects and Classes



A class
(the concept)

Bank
Account

An object (the
realization)

John’s Bank Account
Balance: $5,257

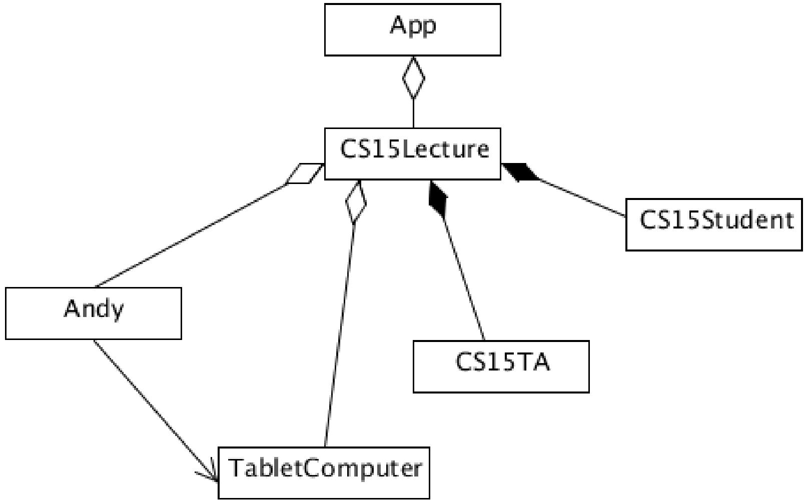
Multiple objects
from the sameclass

Bill’s Bank Account
Balance: $1,245,069

Mary’s Bank Account
Balance: $16,833

OOP Modelling: Program

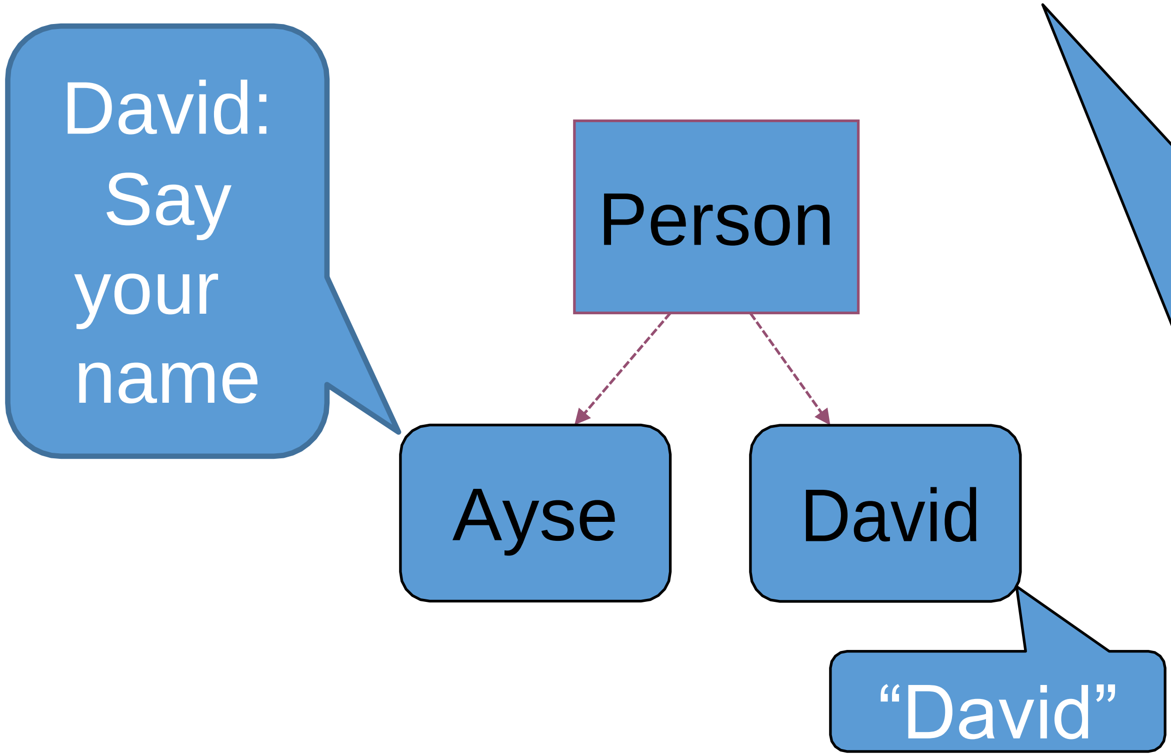
- We write programs by modeling problem as set of **collaborating components**:
    - We determine what the building blocks are
    - Put them together so they cooperate properly
    - Like building with smart Legos, some of which are pre-defined, some of which we design!



OOP Modelling: Program

- Program/Software System
  - Set of objects
  - Which interact with each other

Created (instantiated) from class definitions

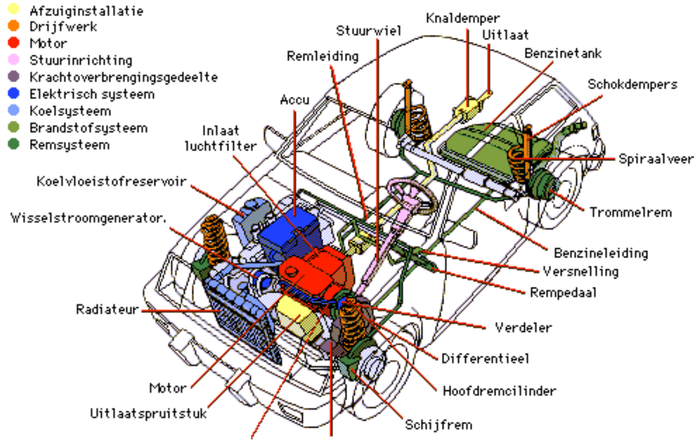


One object will send a message to another object asking it to do a particular task. The first object does not need to know how the task is done (only how to request that it be done.)

This corresponds to calling one of the second object’s methods!

Abstraction

- An abstraction hides (or ignores) unnecessary details
- Denotes the essential properties of an object
- One of the fundamental ways in which we handle complexity
- Objects are abstractions of real world entities
- Programming goal: choose the right abstractions



Microsoft illustratie

Abstraction

A car consists of four wheels an engine, a steering wheel and brakes.

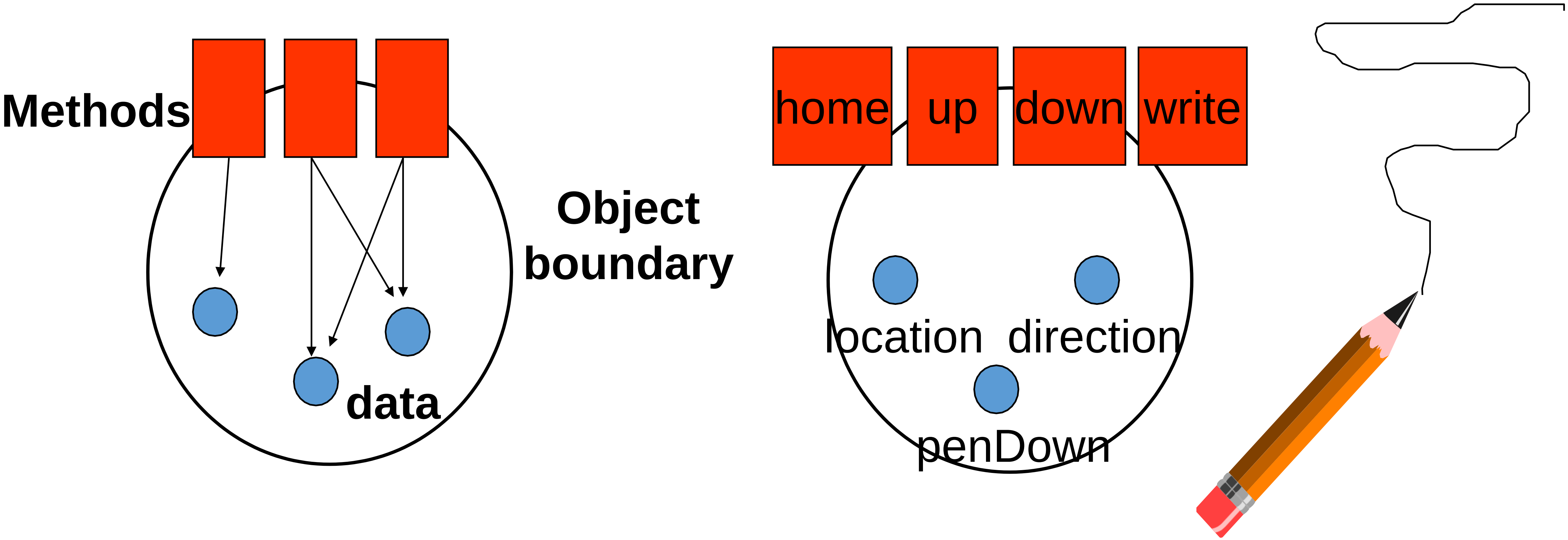
Choosing Abstraction

- Abstractions can be about
  - Tangible things (a vehicle, a car, a map) or
  - Intangible things (a meeting, a route, a schedule)

Example:

- Abstraction name: Lamp
  - Attribute: Wattage (i.e. energy usage)
  - Attribute: On/Off
- There are other possible properties (shape, color, socket size, etc.), but we have decided those are less essential
- The essential properties are determined by the problem

Example



Methods

Object boundary

data

home

up

down

write

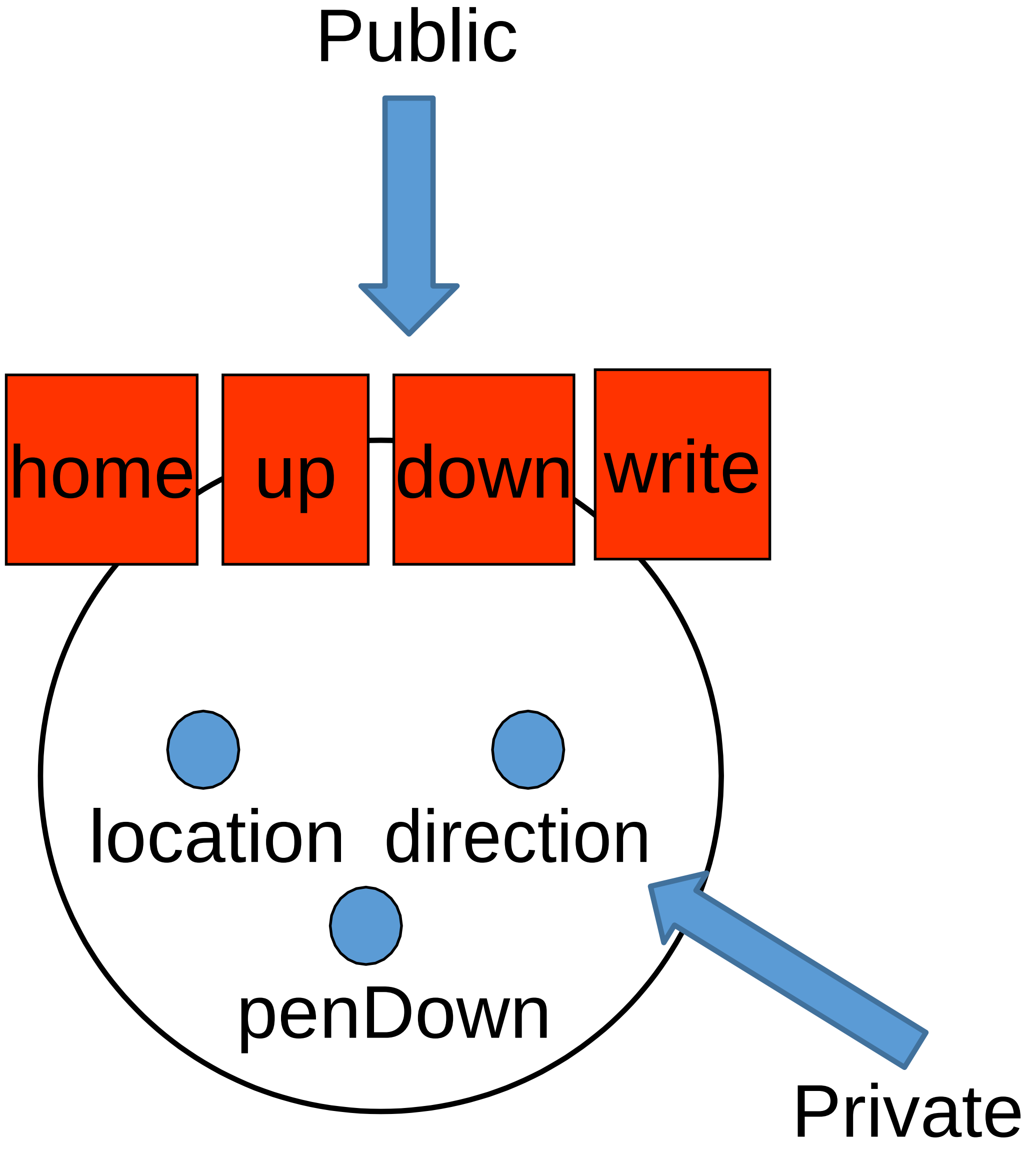
location

direction

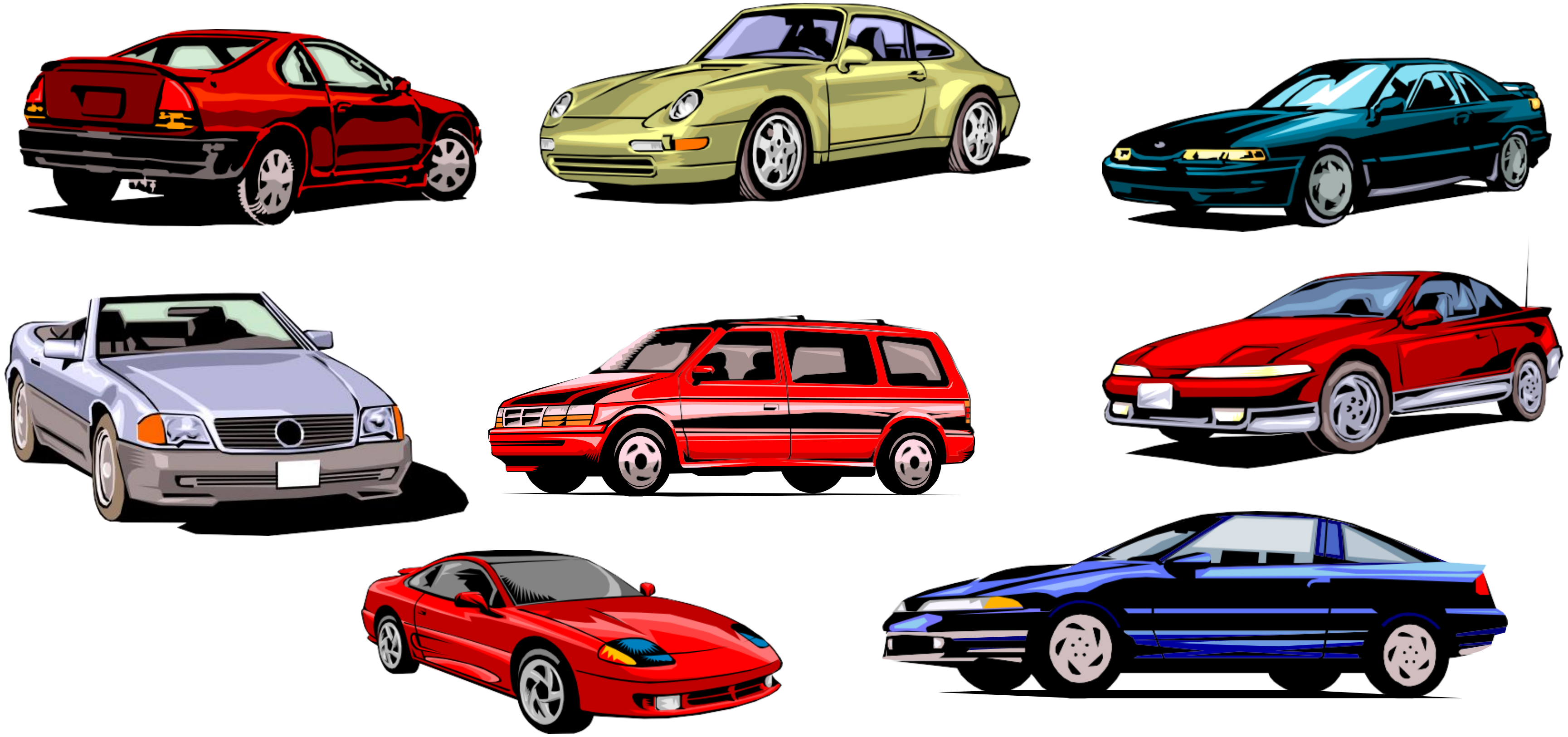
penDown

Encapsulation

- The data belonging to an object is hidden, so variables are private
- Methods are public
- We use the public methods to change or access the private data.
- No dependence on implementation
- Encapsulation makes programming easier
  - As long as the contract is the same, the client doesn't care about the implementation



Creating Objects in Java

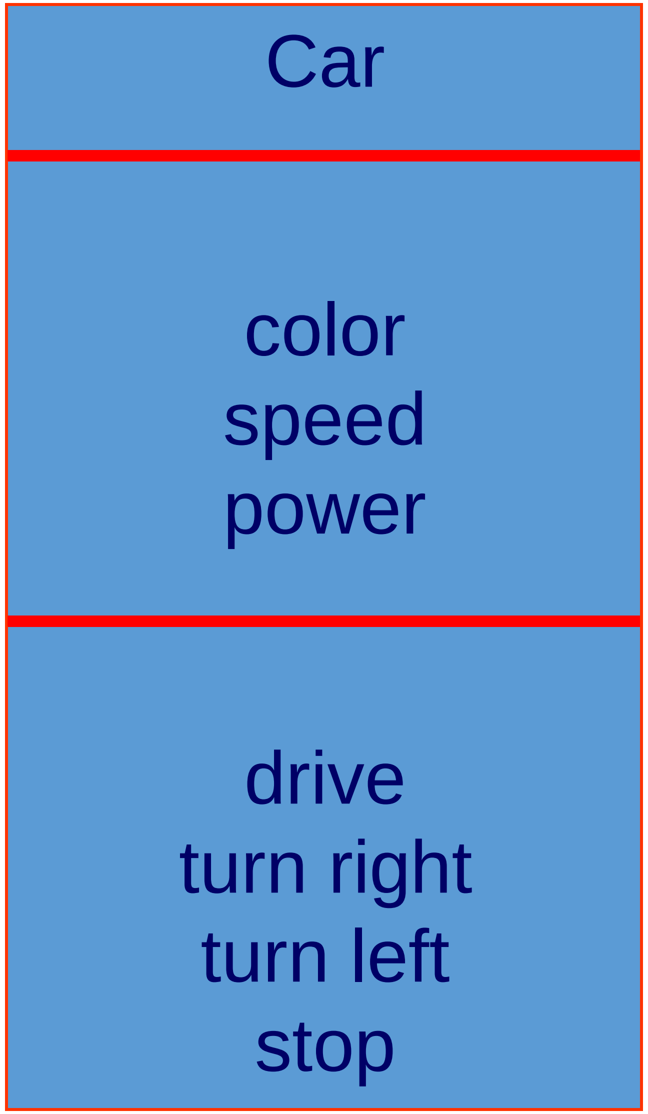


Defining Car Class

- What are the common attributes of cars?

- What are the common behaviors of cars?

Class Car



Car

color
speed
power

drive
turn right
turn left
stop

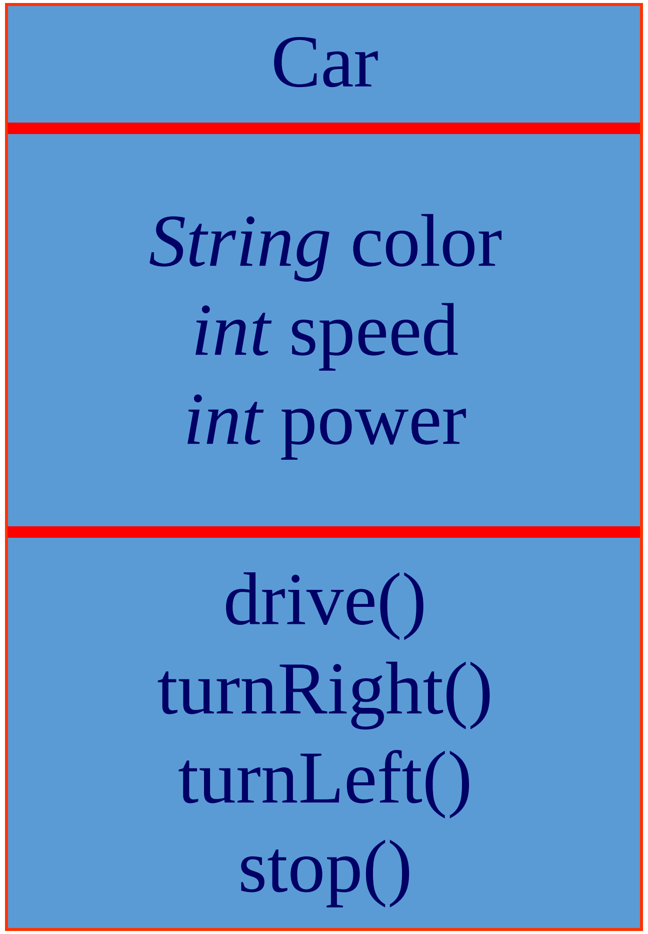
class name

attributes

operations

Java Syntax

```
public class Car
{
// attribute declarations
        private String color;
        private int speed;
        private int power;
// method declarations
   public void drive()
   {  // ….
   }
   public void turnRight()
   {  // ….
   }
}
```

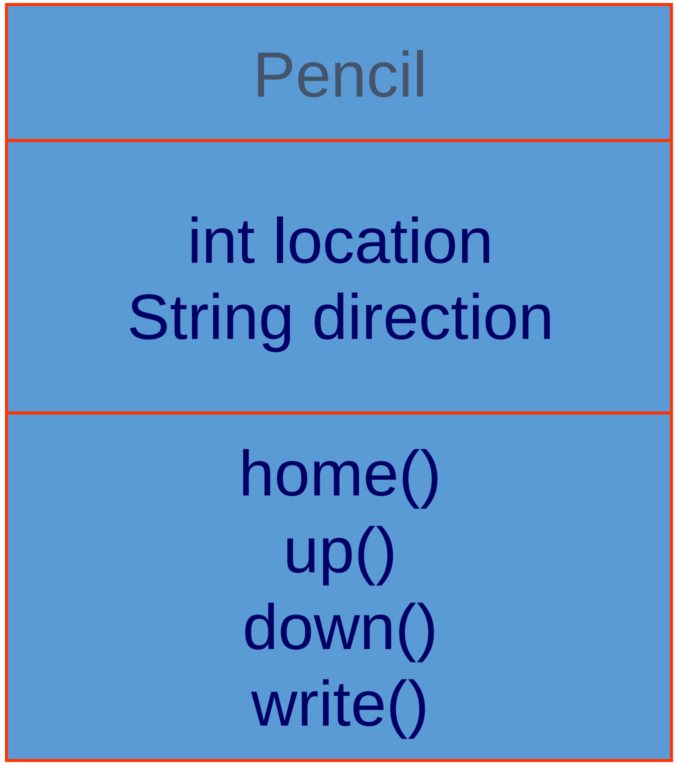


Car

String color
int speed
int power

drive()
turnRight()
turnLeft()
stop()

Class Pencil



Pencil

int location
String direction

home()
up()
down()
write()

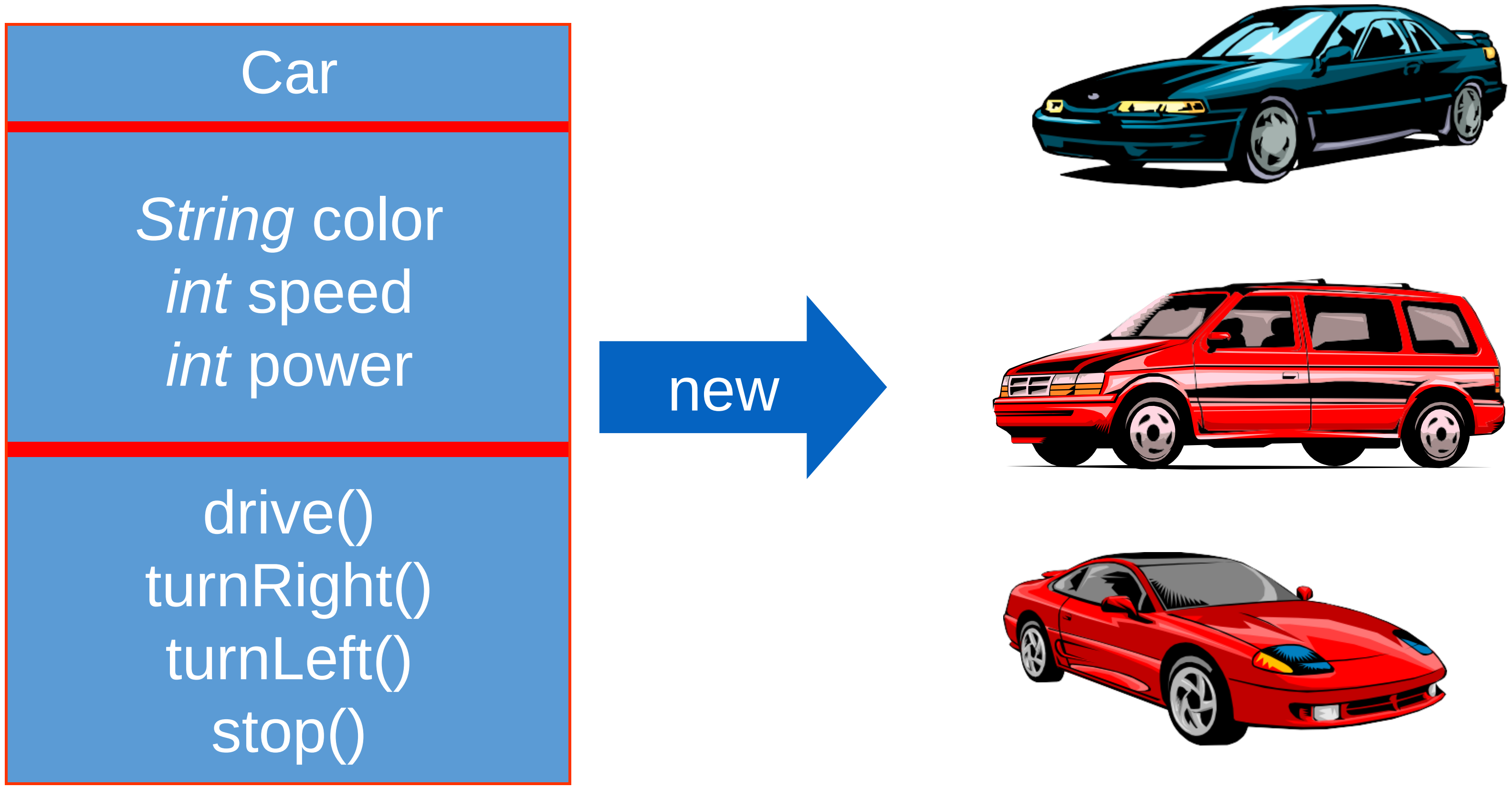
Name

attributes

methods

Declaring objects

- A class can be used to *create* objects
- Objects are the instances of that class



Car

String color
int speed
int power

drive()
turnRight()
turnLeft()
stop()

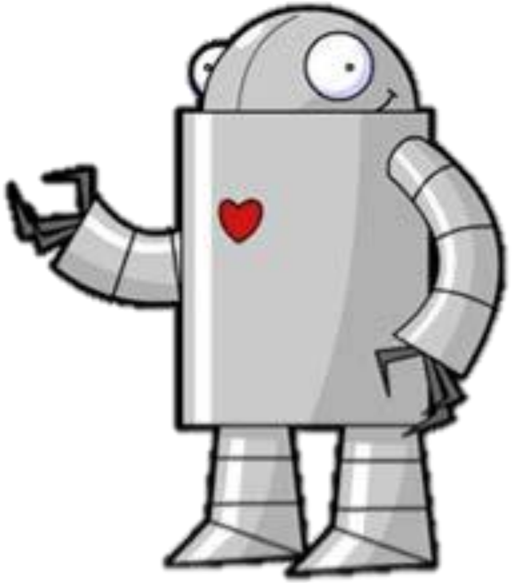
new

Defining and Calling Methods on Objects

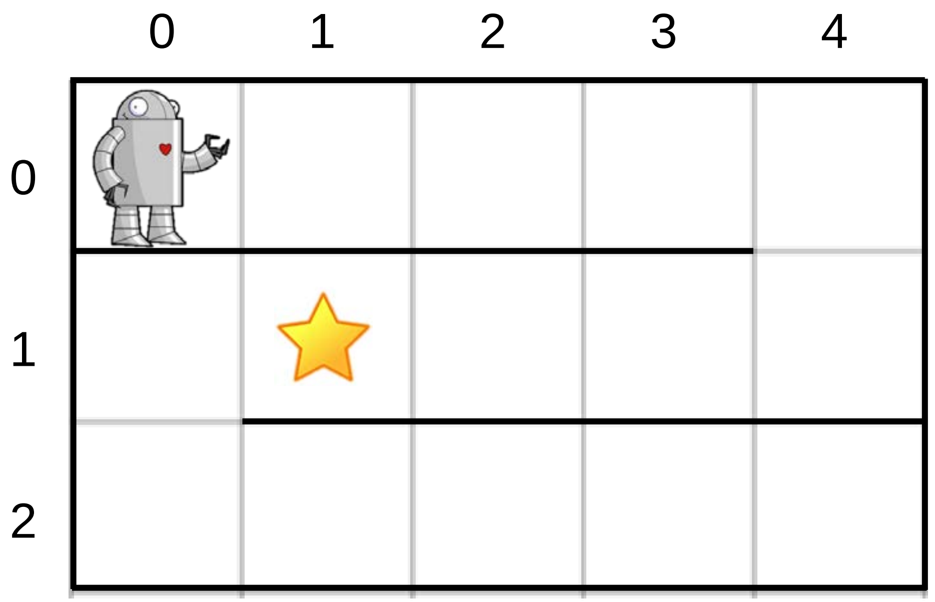
- Calling methods
- Declaring and defining a class
- Instances of a class
- Defining methods
- The this keyword

Meet samBot

- samBot is a robot who lives in a 2D grid world
- He knows how to do two things:
  - move forward any number of steps
  - turn right 90^o
- We will learn how to communicate with samBot using Java



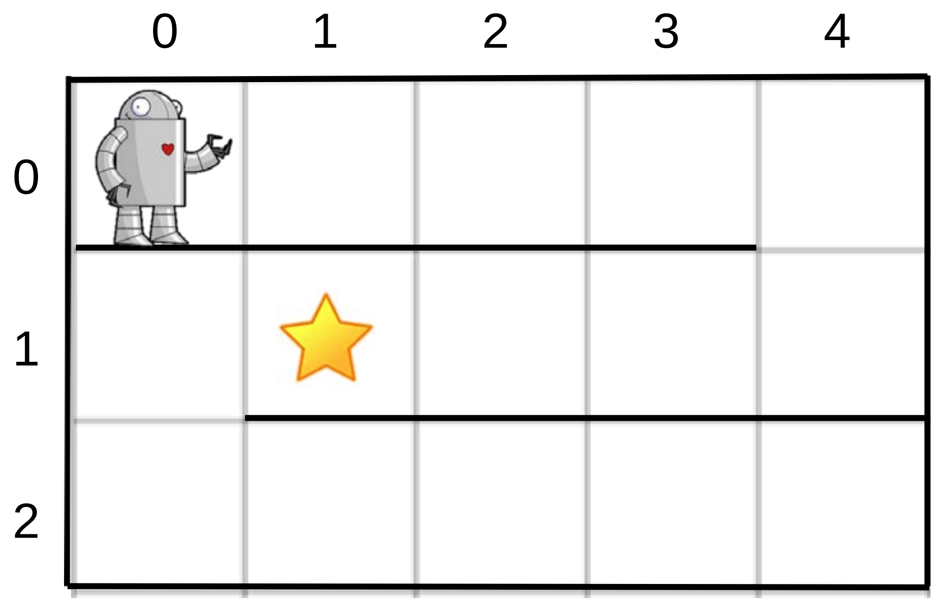
samBot’s World



- This is samBot’s world
- samBot starts in the square at (0,0)
- He wants to get to the square at (1,1)
- Thick black lines are walls that samBot can’t pass through

Giving Instructions

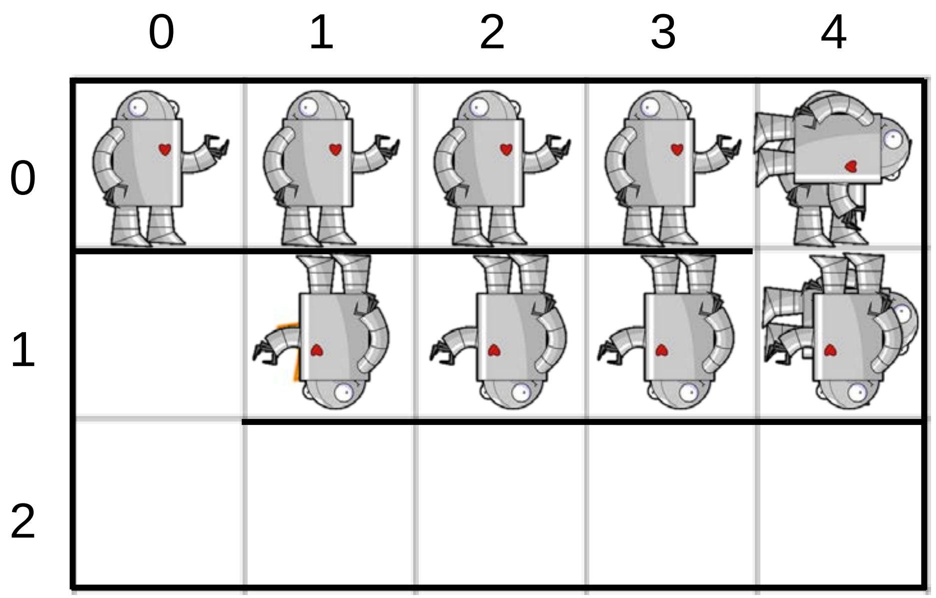
- **Goal**: move samBot from his starting position to his destination by giving him a list of instructions
- samBot only knows instructions “move forward n steps” and “turn right”
- What instructions should we give him?



Giving Instructions

Note: samBot moves in the direction her outstretched arm is pointing; yes, he can move upside down in this 2D world

- “Move forward 4 steps.”
- “Turn right.”
- “Move forward 1 step.”
- “Turn right.”
- “Move forward 3 steps.”

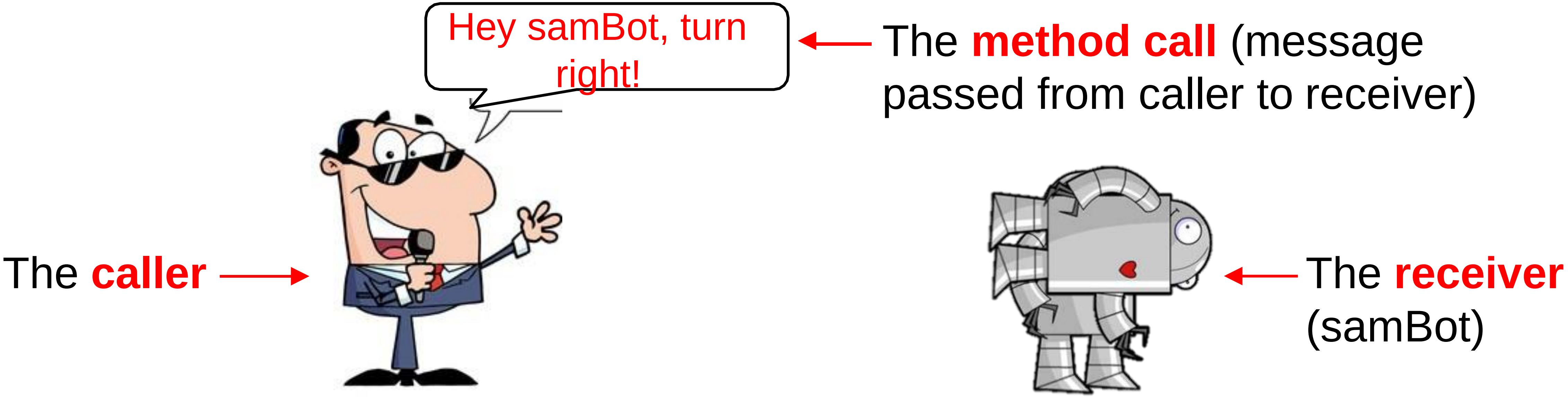


“Calling Methods”: Sending Messages in Java

- samBot can only handle messages that he knows how to respond to
- These responses are called **methods!**
  - “method” is short for “method for responding to a message”
- Objects cooperate by sending each other messages.
  - object sending message is the **caller**
  - object receiving message is the **receiver**

“Calling Methods”: Sending Messages in Java

- samBot already has one method for “move forward n steps” and another method for “turn right”
- When we send a message to samBot to “move forward” or “turn right” in Java, we are **calling a method on samBot.**



Hey samBot, turn right!

The **method call** (message passed from caller to receiver)

The **caller**

The **receiver** (samBot)

Turning samBot right

- samBot's “turn right” method is called turnRight
- To call the turnRight method on samBot:
- samBot.turnRight();
- To call methods on samBot in Java, need to address him by name!
- Every command to samBot takes the form:
- samBot.<method name(...)>;

You substitute for anything in < >!

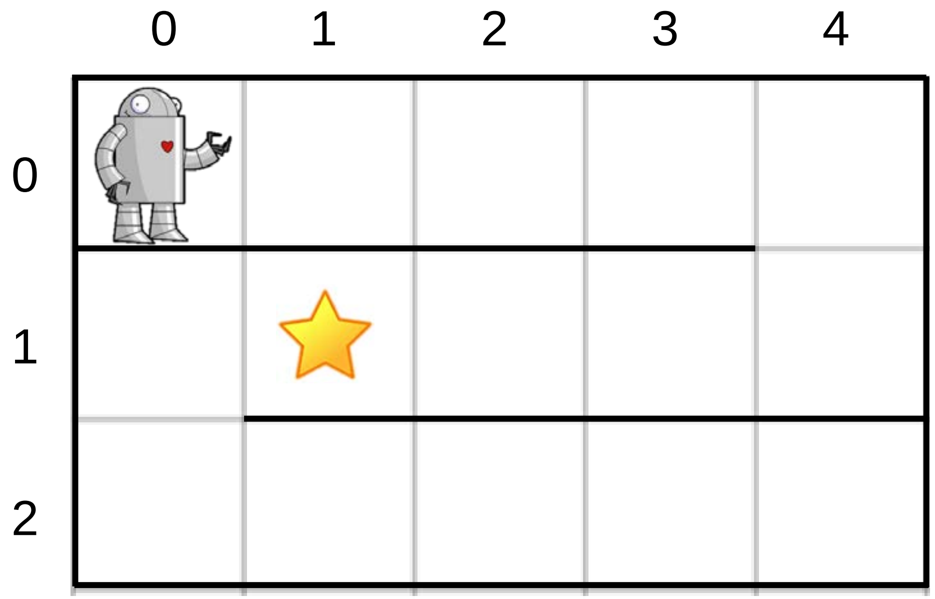
; ends Java statement

- What are those parentheses at the end of the method for?

Guiding samBot in Java

- Tell samBot to move forward 4 steps → `samBot.moveForward(4);`
- Tell samBot to turn right → `samBot.turnRight();`
- Tell samBot to move forward 1 step → `samBot.moveForward(1);`
- Tell samBot to turn right → `samBot.turnRight();`
- Tell samBot to move forward 3 steps → `samBot.moveForward(3);`

“pseudocode”



Java code

Putting Code Fragment in a Real Program

- Let’s demonstrate this code for real
- First, need to put it inside real Java program
- Grayed-out code specifies context in which samBot executes these instructions
  - Also includes samBot’s capability to respond to moveForward and turnRight – more on this later

```
•public class RobotMover {

    /* additional code */

    public void moveRobot(Robot samBot) {
        samBot.moveForward(4);
        samBot.turnRight();
        samBot.moveForward(1);
        samBot.turnRight();
        samBot.moveForward(3);
    }
}
```

Putting Code Fragments in a Real Program

Now we will explain this part of the code.

```
public class RobotMover {

    /* additional code elided */

    public void moveRobot(Robot samBot) {
        samBot.moveForward(4);
        samBot.turnRight();
        samBot.moveForward(1);
        samBot.turnRight();
        samBot.moveForward(3);
    }
}
```

- Before, we’ve talked about objects that handle messages with "methods"

Class (refresh)

- A **class** is a blueprint for a certain type of object
- An object’s class defines its properties and capabilities (methods)
- So far, we’ve been working within the class RobotMover
- We need to tell Java about our RobotMover

```
public class RobotMover {

    /* additional code elided */

    public void moveRobot(Robot samBot) {
        samBot.moveForward(4);
        samBot.turnRight();
        samBot.moveForward(1);
        samBot.turnRight();
        samBot.moveForward(3);
    }
}
```

Declaring and Defining a Class (1/3)

- As with dictionary entry, first **declare** term, then provide **definition**
- First line **declares** RobotMover class
- Breaking it down:
  - public indicates that anyone can use this class
  - class indicates to Java that we are about to define a new class
  - RobotMover is the name that we have chosen for our class

declaration of the RobotMover class

```
public class RobotMover {

        /* additional code elided */

        public void moveRobot(Robot samBot){
                samBot.moveForward(4);
                samBot.turnRight();
                samBot.moveForward(1);
                samBot.turnRight();
                samBot.moveForward(3);
         }
}
```

Note: public and class are Java “reserved words” aka “keywords” and have pre-defined meanings in Java; we’ll be using Java keywords a lot in the future

Declaring and Defining a Class (2/3)

- **Class definition** (aka “body”) defines properties and capabilities of class
    - it is contained within curly braces that follow the class declaration
- A class’s capabilities (“what it knows how to do”) are defined by its **methods –** RobotMover thus far only knows this very specific moveRobot method
- A class’s properties are defined by its **instance variables** – more on this next week

```
•public class RobotMover {

    •/* additional code elided */

    •public void moveRobot(Robot samBot){
•           samBot.moveForward(4);
•           samBot.turnRight();
•           samBot.moveForward(1);
•           samBot.turnRight();
•           samBot.moveForward(3);
    •}

•}
```

definition of the RobotMover class

Declaring and Defining a Class (3/3)

- **General form for a class:**

```
<visibility> class <name> {

   <code (properties and
capabilities) that defines class>


}
```

declaration

definition

- Each class goes in its own file, where name of file matches name of class
  - RobotMover class is contained in file “RobotMover.java”

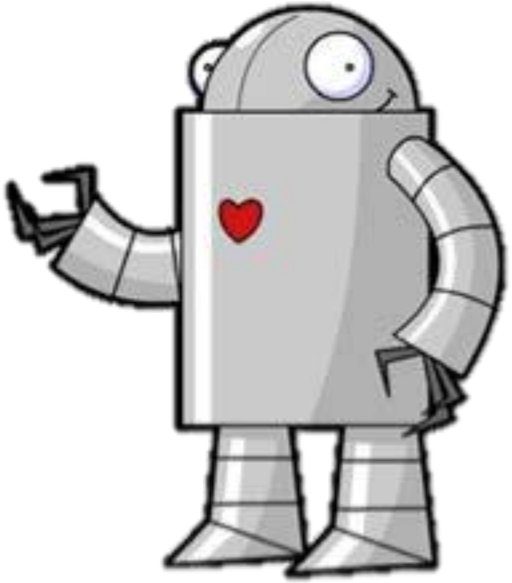
Methods of the Robot class

```
public class Robot {

        public void turnRight() {
            // code that turns robot right
        }

        public void moveForward(int numberOfSteps) {
            // code that moves robot forward
        }

        /* other code deleted */
}
```

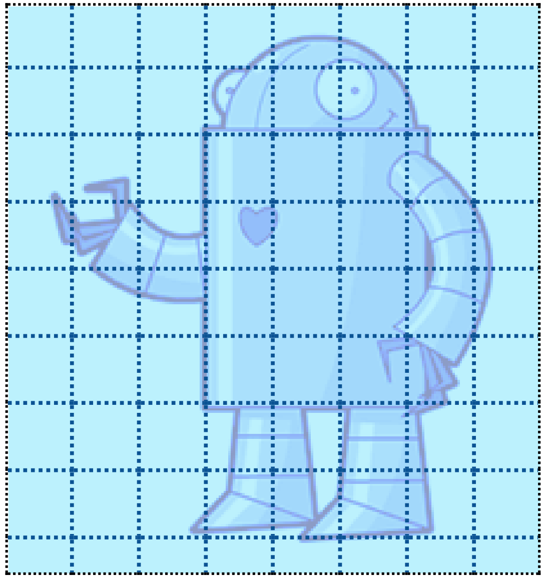
- public void turnRight() and public void moveForward(int numberOfSteps) each **declare a method**
- Since moveForward needs to know how many steps to move, we put int numberOfSteps within the parentheses
    - int is Java’s way of saying this parameter is an “integer” (we say “of type integer”)

Classes and Instances (1/3)

- We've been saying samBot is a Robot
- We'll now refer to him as an **instance** of class Robot
  - This means samBot is a particular Robot built using Robot class as a blueprint
- All Robots (all **instances** of the class Robot) have the exact same capabilities: the methods defined in the Robot class



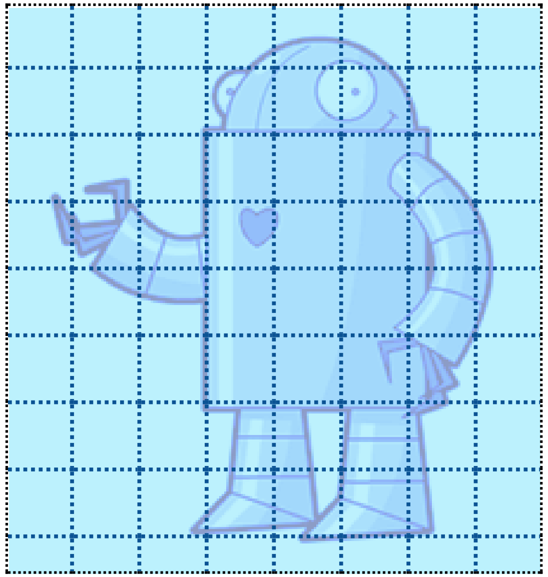
Classes and Instances (2/3)



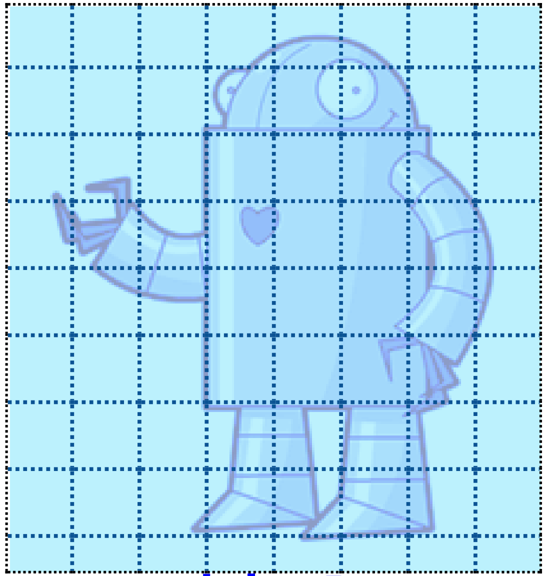
The Robot class is
like a blueprint

Classes and Instances (3/3)

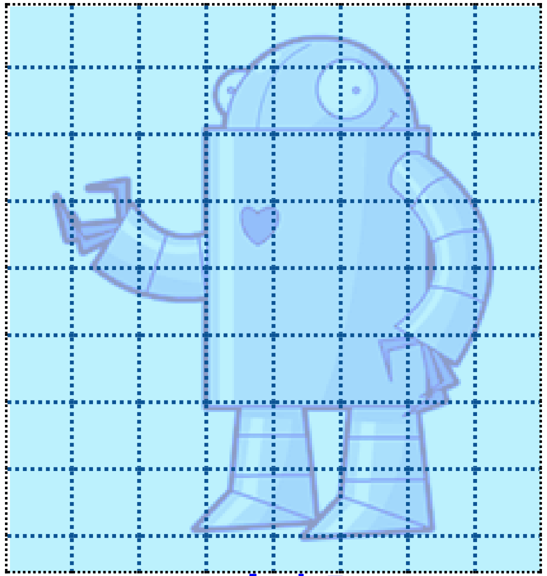
We can use the Robot class to build actual Robots - **instances** of the class Robot, whose properties may vary (next lecture)



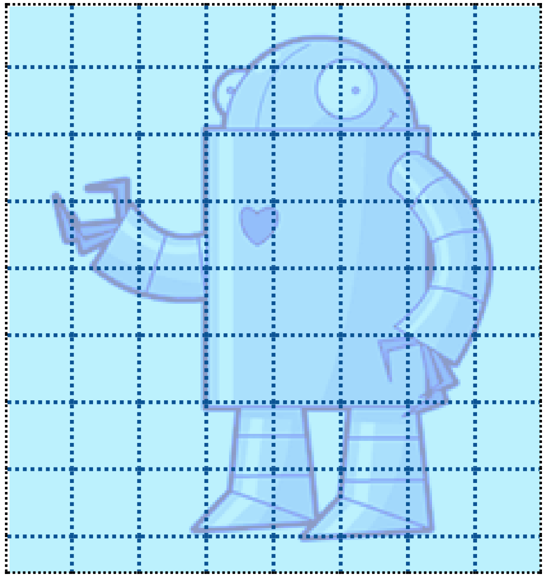
samBot



blueBot



pinkBot

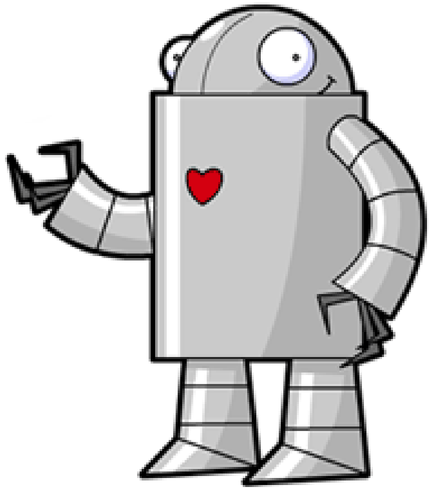


greenBot

Classes and Instances

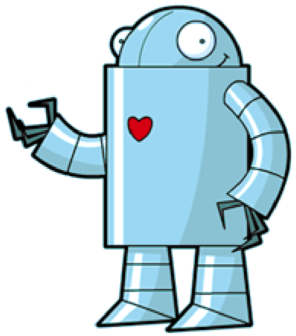
Method calls are done on instances of the class

instance



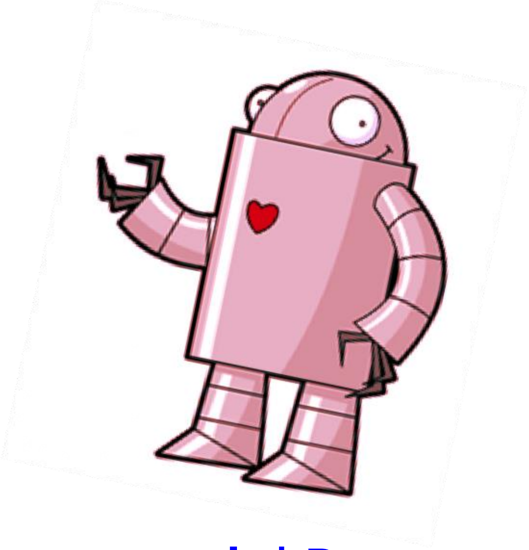
samBot

instance



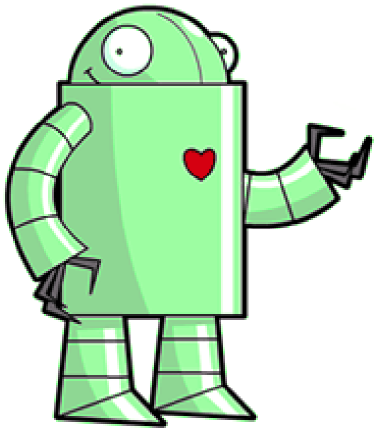
blueBot

instance



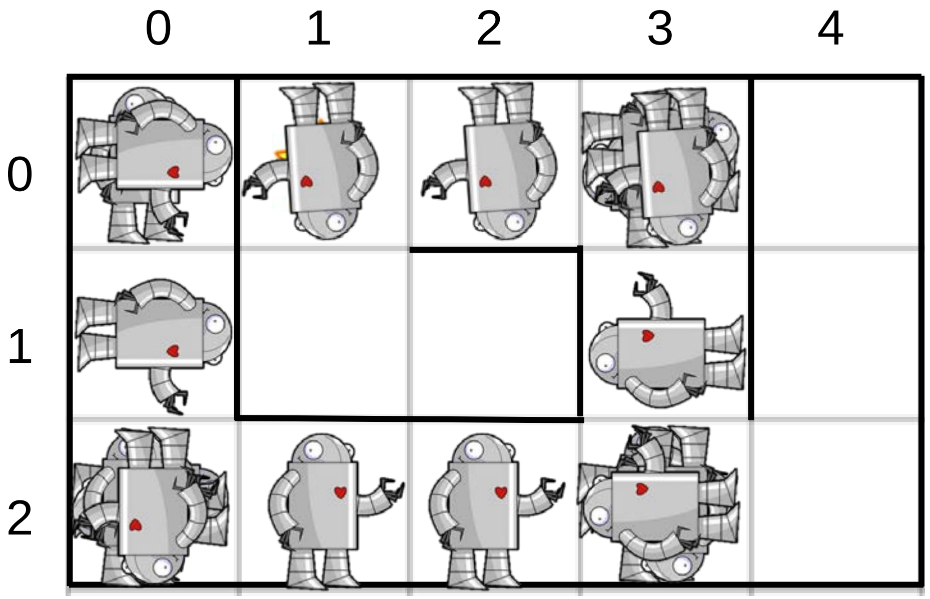
pinkBot

instance



greenBot

A variation



```
public class RobotMover {
     /* additional code elided */

     public void moveRobot(Robot samBot){
          samBot.turnRight();
             samBot.moveForward(2);
             samBot.turnRight();
             samBot.turnRight();
             samBot.turnRight();
             samBot.moveForward(3);
             samBot.turnRight();
             samBot.turnRight();
             samBot.turnRight();
             samBot.moveForward(2);
             samBot.turnRight();
             samBot.turnRight();
             samBot.turnRight();
             samBot.moveForward(2);
       }
}
```

A variation

- Lots of code for a simple problem...
- samBot only knows how to turn right, so have to call turnRight three times to make him turn left
- If he understood how to “turn left”, would be much simpler!
- We can modify samBot to turn left by **defining a method** called turnLeft

```
public class RobotMover {
    /* additional code elided */

    public void moveRobot(Robot samBot){
        samBot.turnRight();
            samBot.moveForward(2);
            samBot.turnRight();
            samBot.turnRight();
            samBot.turnRight();
            samBot.moveForward(3);
            samBot.turnRight();
            samBot.turnRight();
            samBot.turnRight();
            samBot.moveForward(2);
            samBot.turnRight();
            samBot.turnRight();
            samBot.turnRight();
            samBot.moveForward(2);
    }
}
```

Defining a Method (2/2)

```
public class Robot {

    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    public void turnLeft() {
        //The new code goes here!!


  }
}
```

- Adding a new method: turnLeft
- To make a Robot turn left, tell her to turn right three times

The this keyword (1/2)

```
public class Robot {

         public void turnRight() {
            // code that turns robot right
        }

        public void moveForward(int numberOfSteps) {
            // code that moves robot forward
        }

        public void turnLeft() {
            this.turnRight();
            this.turnRight();
            this.turnRight();
        }
}
```

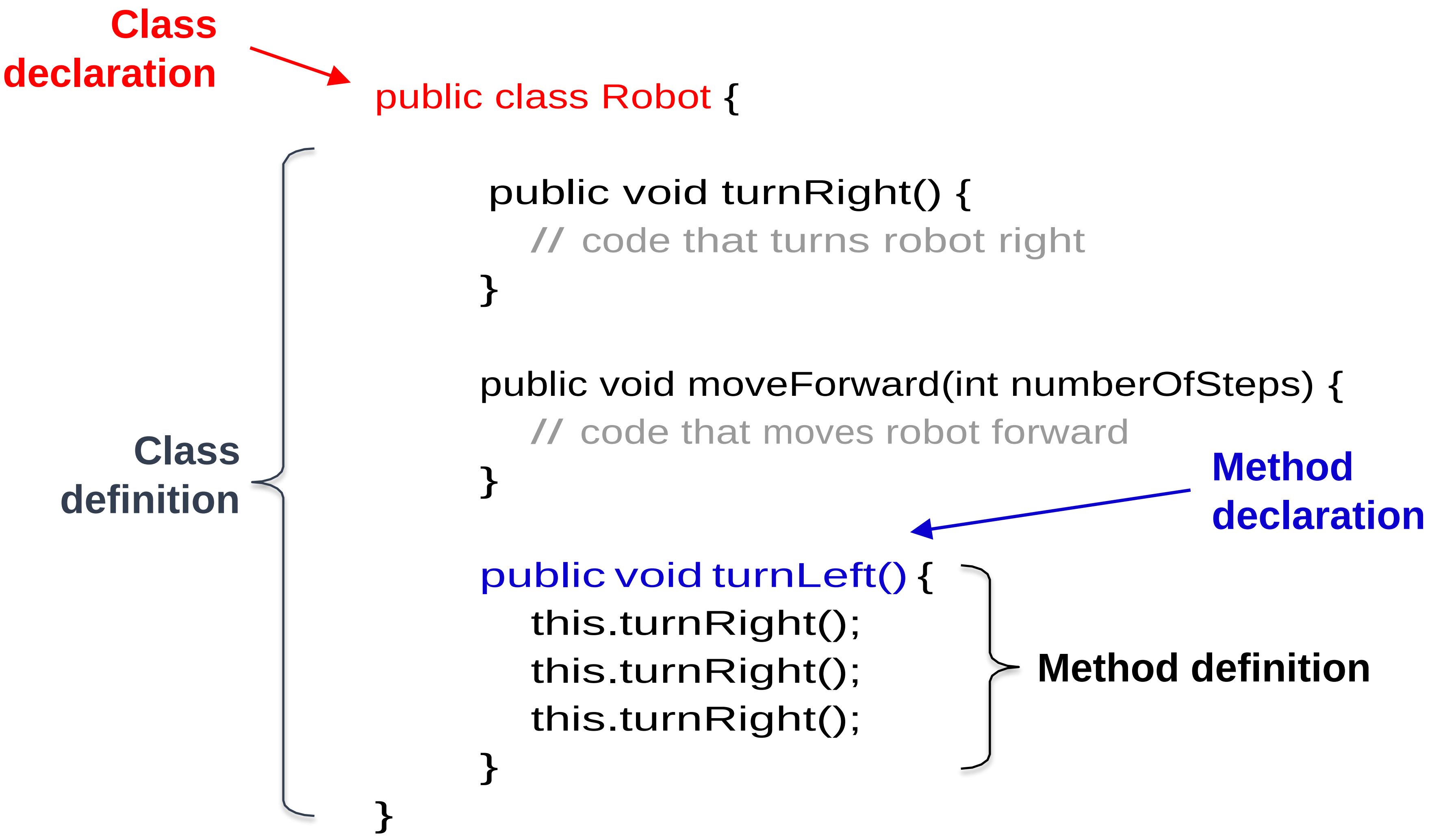
- When working with RobotMover, we were talking to samBot, an instance of class Robot
- To tell her to turn right, we said “samBot.turnRight();”
- Why do we now write “this.turnRight();”?

The this keyword (2/2)

```
•public class Robot {

        public void turnRight() {
•           // code that turns robot right
        }

        public void moveForward(int numberOfSteps) {
•           // code that moves robot forward
        }

        •public void turnLeft() {
•           this.turnRight();
•           this.turnRight();
•           this.turnRight();
        •}
}
```

- The this keyword is how an instance (like samBot) can call a method on itself
- Use this to call a method of Robot class from within another method of Robot class
- When samBot is told by, say, RobotMover to turnLeft, she responds by telling herself to turnRight three times
- this.turnRight(); means “hey me, turn right!”
- this is optional, but desirable!

Summary



Class declaration

Class definition

Method declaration

Method definition

```
public class Robot {

    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }
}
```

Simplifying our code using turnLeft

```
public class RobotMover {
    public void moveRobot(Robot samBot){
      samBot.turnRight();
        samBot.moveForward(2);
        samBot.turnRight();
        samBot.turnRight();
        samBot.turnRight();
        samBot.moveForward(3);
        samBot.turnRight();
        samBot.turnRight();
        samBot.turnRight();
        samBot.moveForward(2);
        samBot.turnRight();
        samBot.turnRight();
        samBot.turnRight();
        samBot.moveForward(2);
    }
}
```

```
public class RobotMover {
    public void moveRobot(Robot samBot){
      samBot.turnRight();
        samBot.moveForward(2);
        •samBot.turnLeft();
        samBot.moveForward(3);
        •samBot.turnLeft();
        samBot.moveForward(2);
        •samBot.turnLeft();
        samBot.moveForward(2);
    }
}
```

We’ve saved a lot of lines of code by using turnLeft!

turnAround

- We could also define a method that turns the Robot around 180º.
- Excercise: Can you declare and define the method **turnAround**

```
public class Robot {
        public void turnRight()  {
             // code that turns robot  right
         }

         public void moveForward(int numberOfSteps) {
              // code that moves robot forward
         }

         public void turnLeft() {
              this.turnRight();
              this.turnRight();
              this.turnRight();
         }

         // your code goes here!
         // ...
         // ...
         // ...
}
```

turnAround

- Now that the Robot class has the method turnAround, we can call the method on any Robot
- There are other ways of implementing this method that are just as correct

```
•public class Robot {
    public void turnRight()  {
        // code that turns robot  right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    •}

    public void turnAround() {
        this.turnRight();
        this.turnRight();
    }
}
```

turnAround

- Instead of calling turnRight, could call our newly created method, turnLeft
- Both of these solutions are equally correct, in that they will turn the robot around 180°
- How do they differ? When we try each of these implementations with samBot, what will we see in each case?

```
public class Robot {
    public void turnRight()  {
        // code that turns robot  right
    }

    public void moveForward(int numberOfSteps)  {
        // code that moves robot forward
    }

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }

    public void turnAround() {
        this.turnLeft();
        this.turnLeft();
    }
}
```

Java Operator Precedence and Associativity

Operators	**Precedence**	**Associativity**
Postfix increment and decrement	++ --	left to right
Prefix increment and decrement, and unary	++ -- + - ~ !	right to left
Multiplicative	* / %	left to right
Additive	+ -	left to right
Shift	<< >> >>>	left to right
Relational	< > <= >= instanceof	left to right
Equality	== !=	left to right
Bitwise AND	&	left to right
Bitwise exclusive OR	^	left to right
Bitwise inclusive OR	|	left to right
Logical AND	&&	left to right
Logical OR	||	left to right
Ternary	? :	right to left
Assignment	= += -= *= /= %= &= ^= |= <<=>>= >>>=	left to right

Increment & Decrement Operator:-

Increment:-

It is used to increment a value by 1. There are two varieties of increment operator:

- **Post-Increment :** Value is first used for computing the result and then incremented.
- **Pre-Increment :** Value is incremented first and then result is computed.

- Ex.-

```
public class Test {
     public static void main(String[] args)
     {
          int a = 10;
          int b = ++a;
          int c=a++;

          // uncomment below line to see error
          // b =  10++;
         //   b=++(++a);

          System.out.println(a,b,c);
     }
}
```

Decrement:-

It is used for decrementing the value by 1. There are two varieties of decrement operator.

- **Post-decrement :** Value is first used for computing the result and then decremented.
- **Pre-decrement :** Value is decremented first and then result is computed.

- Ex.-

```
public class Test {
     public static void main(String[] args)
     {
          int a = 10;
          final int b = a--;
         int c=--b; //error

        // uncomment below line to see error
        // boolean d = false;
       // d++;

          System.out.println(a,b,c);
     }
}
```