Day 01: Introduction to OOP

Slide Credits: *Internet / Chetan Arora*

Methodology

- Module Duration: 2nd – 4th August (3 days module)
- Lectures: Wed - Fri (11:30 am – 1pm)
- Class code : 7opts3f

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

ıre https://www.tiobe.com/tiobe-index/

Jul 2017	Jul 2016	Change	Programming Language	Ratings	Change
1	1		Java	13.774%	-6.03%
2	2		C	7.321%	-4.92%
3	3		C++	5.576%	-0.73%
4	4		Python	3.543%	-0.62%
5	5		C#	3.518%	-0.40%
6	6		PHP	3.093%	-0.18%
7	8	^	Visual Basic .NET	3.050%	+0.53%
8	7	v	JavaScript	2.606%	-0.04%
9	12	^	Delphi/Object Pascal	2.490%	+0.45%
10	55	^^	Go	2.363%	+2.20%
11	9	v	Perl	2.334%	-0.09%

TIOBE Index for July 2017

11 Buzzwords for Java

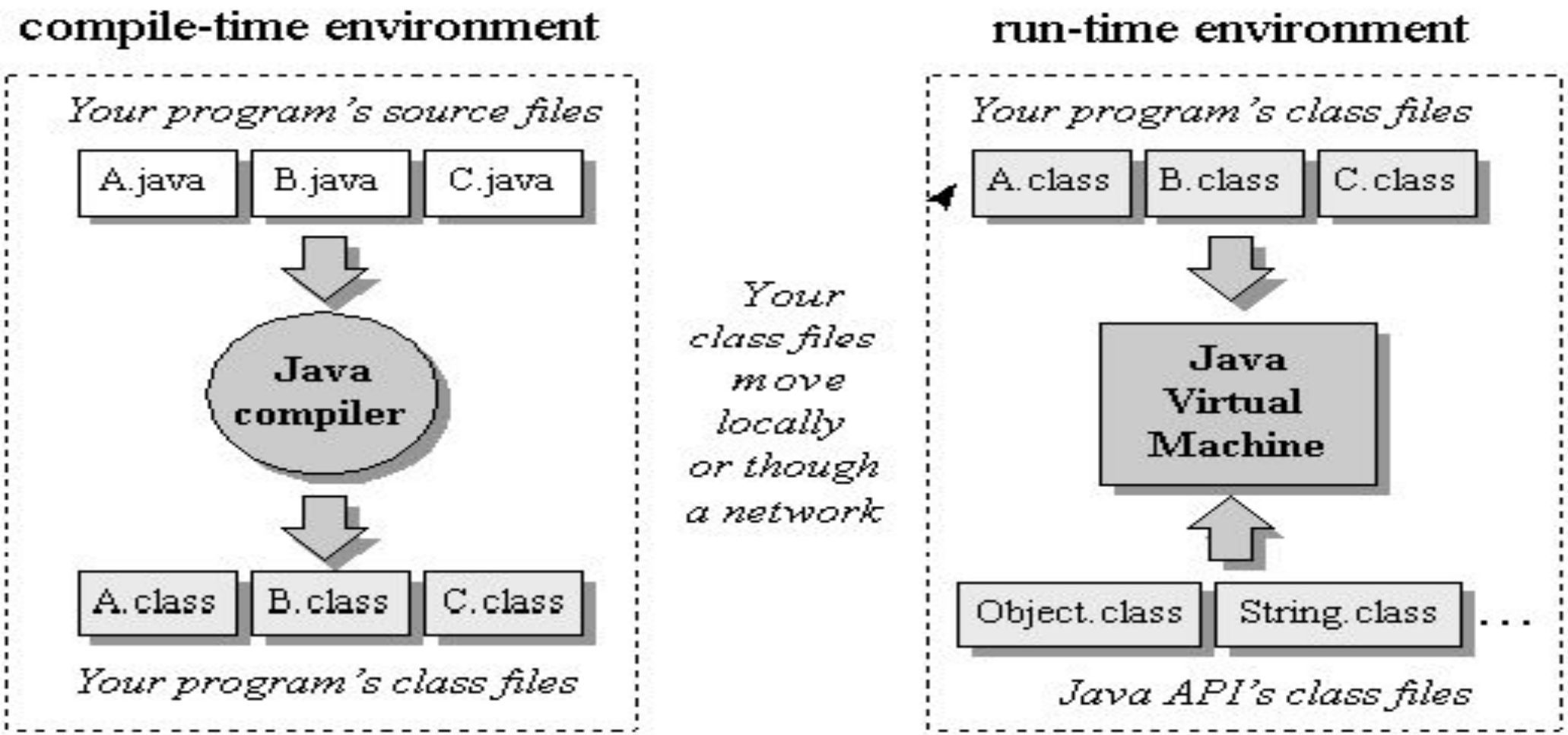
- Simple
- Object Oriented
- Network Savvy
- Robust
- Secure
- Architecture Neutral
- Portable
- Interpreted
- High performance
- Multithreaded
- Dynamic

- Simple : A cleaned up version of C++. No need for pointer arithmetic, structures, unions, virtual base class etc. Also, it's basic functionalities like interpreter, class support, basic libraries take ~ 215K space.
- Object-Oriented : The focus is on the object and on the interfaces of the objects.
- Distributed : It could connect to web servers (HTTP, FTP, TCP/IP)
- Robust : No memory corruption (no explicit pointers, garbage collection, bound checking). Also robust compile time(syntax checking, type checking) and runtime checks (NullPointerException, ArrayIndexOutOfBoundException)
- Secure: It is designed to avoid certain vulnerabilities like Overrunning the runtime stack, corrupting memory outside its own process space, reading or writing files without permission etc.

- Architecture-Neutral : i.e. it can run on different types of computers without any changes to code. It does so by converting any code into byte code, which is not computer specific.
- Portable: Standardization of sizes of primitive data types, their arithmetics etc. (Exception : UI libraries in Java).
- Interpreted: Since linking process is more incremental and lightweight process, the development process can be much more rapid and exploratory.
- High-Performance: just-in-time compiler can sometimes outperform traditional compilers because of availability of more information.
- Multithreaded: Necessary as Moore's Law has come to an end!
- Dynamic: It allows editing an already running code.

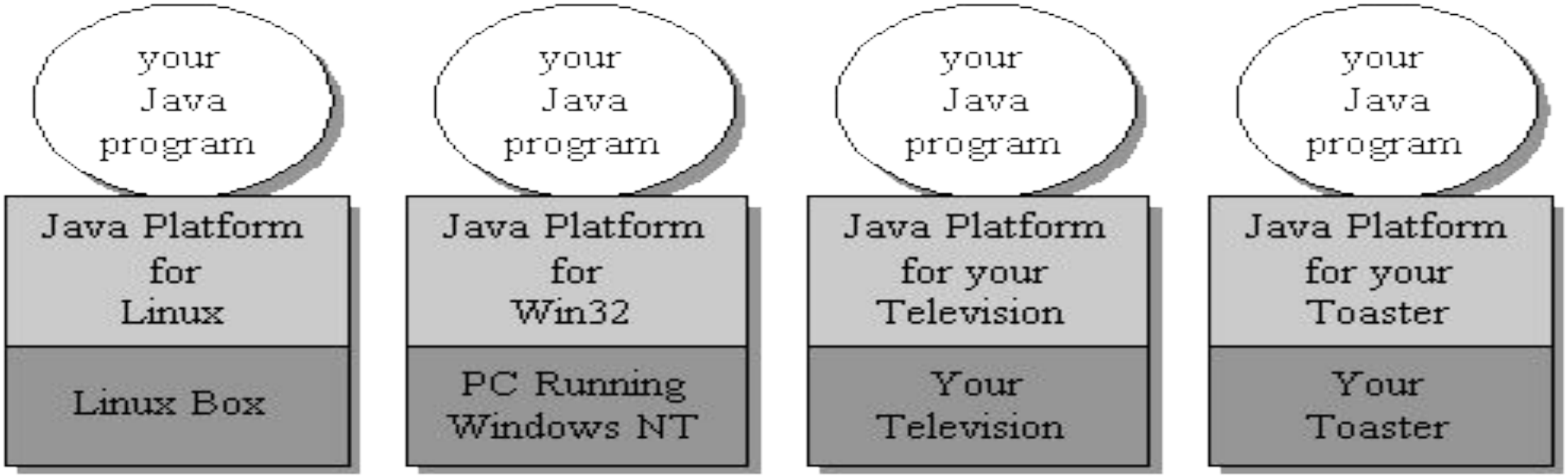
Java Architecture

- Java programming environment



Java Architecture

- Java platform (Java Virtual Machine + Java API)



Structured Programming

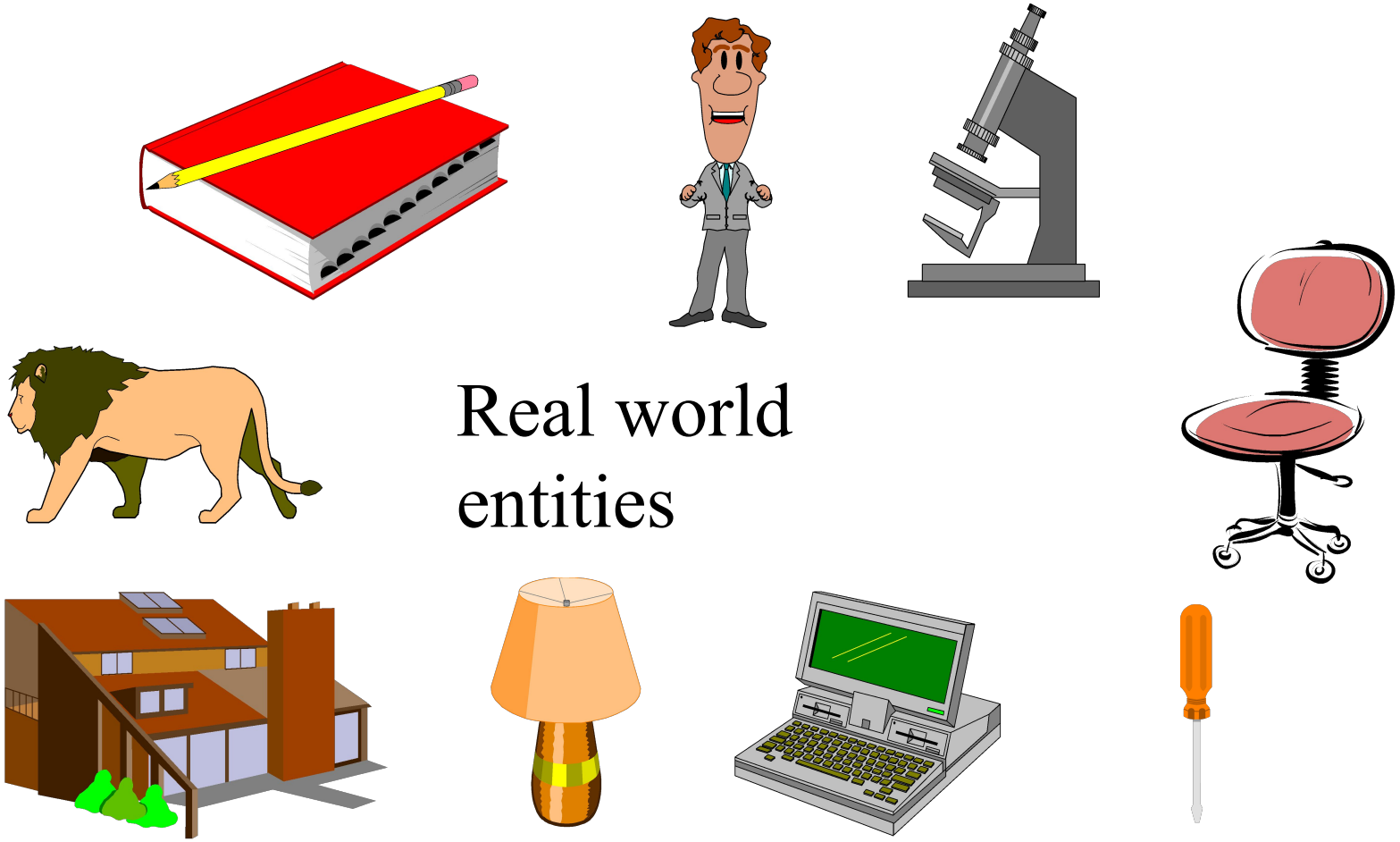
- Methods define the structure of the programs, they are basic building blocks

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

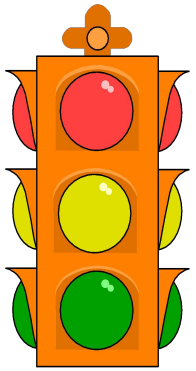
Object Oriented Programming

- 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.

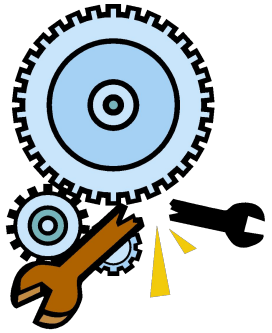
Real World



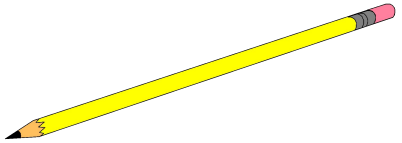
Objects have states



Red

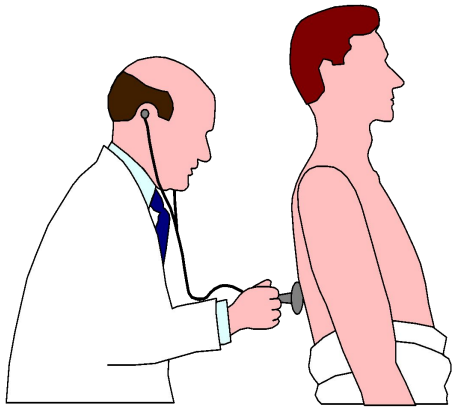


Broken

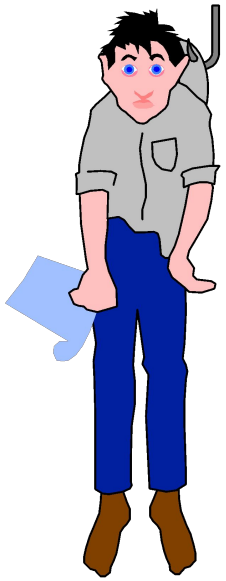


Lyin
g

Happy

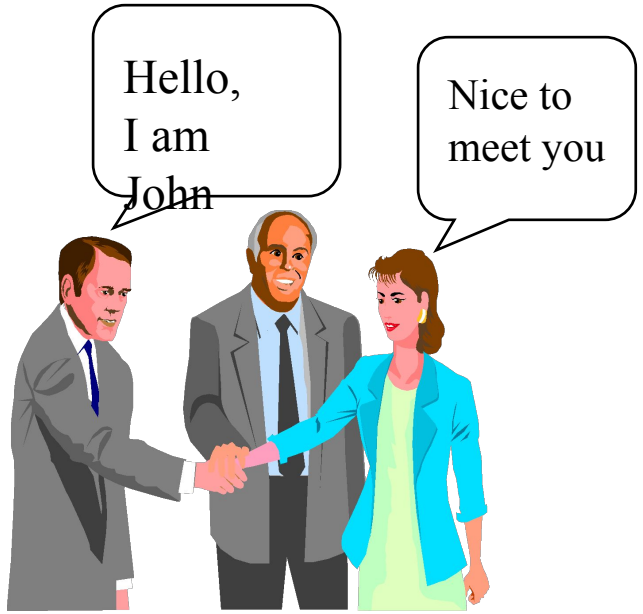


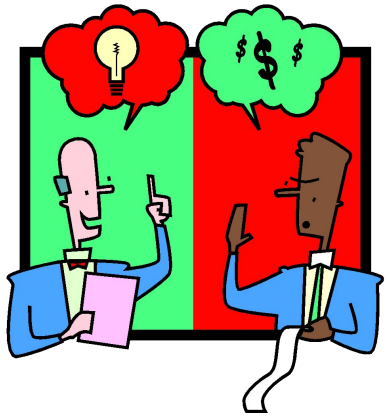
il
l

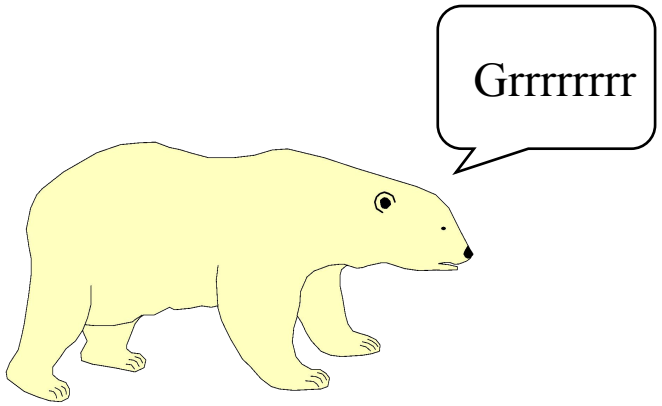


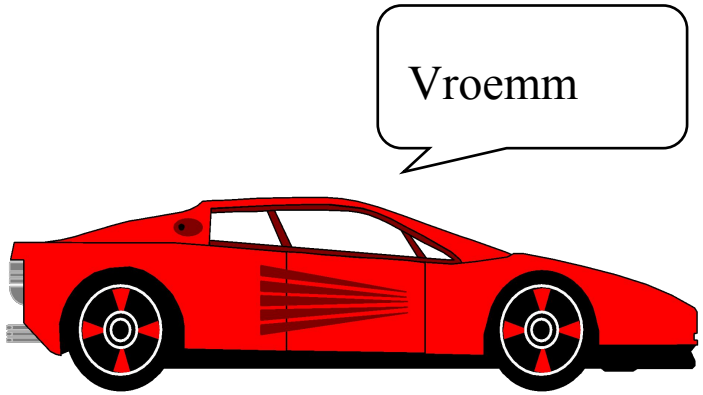
Hooke
d

Objects have behavior









Object Properties

- Identity
- State
- Behavior

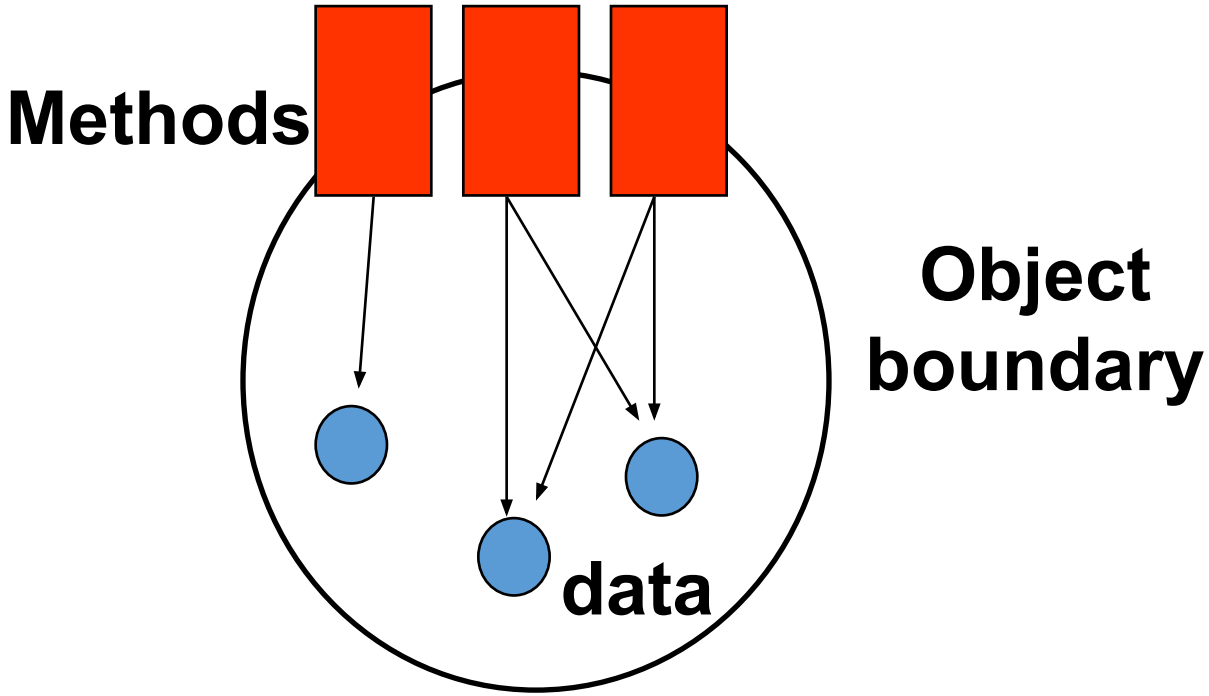
on
off

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)

An object
(the realization)

Multiple objects
from the same class

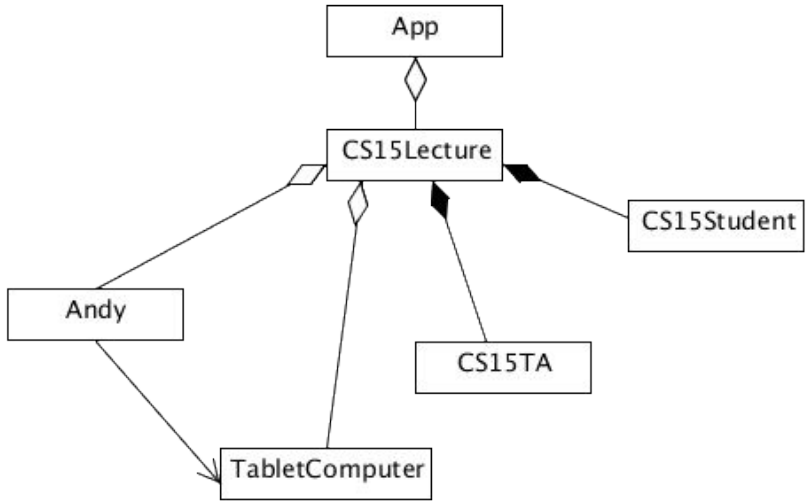
John's Bank Account
Balance: $5,257

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

Mary's Bank Account
Balance: $16,833

OOP Modelling: Program

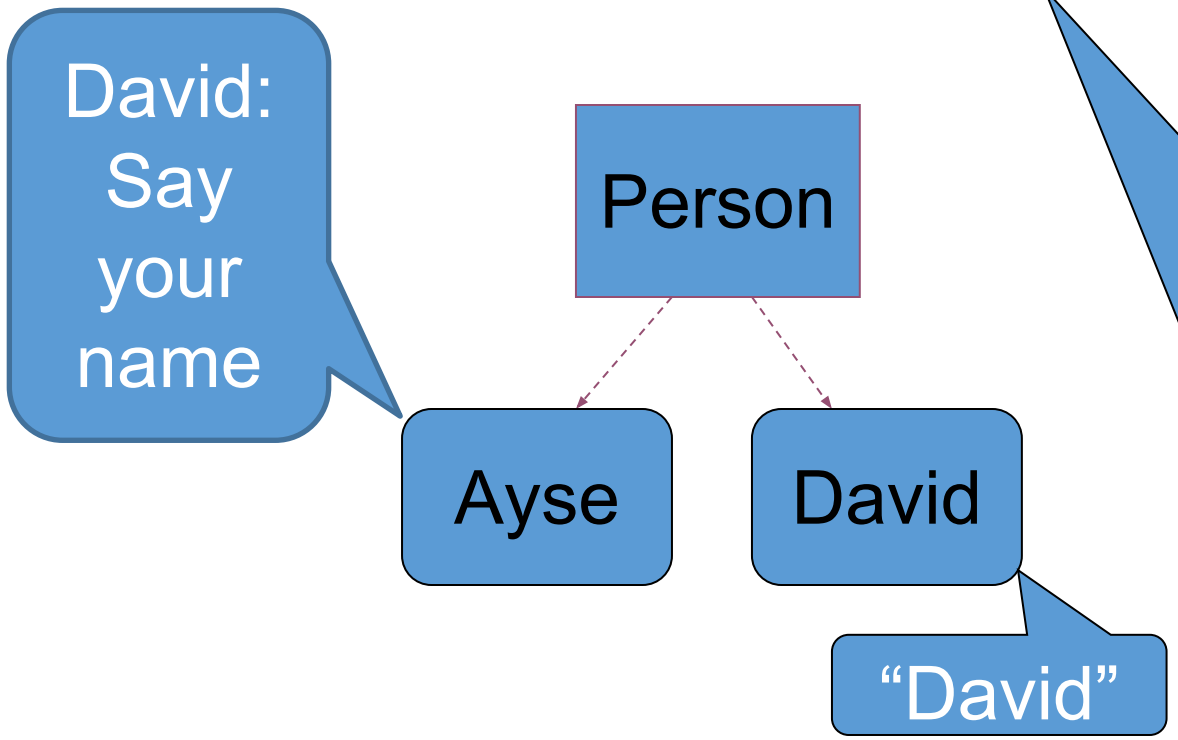
- We write programs by modeling problem as set of **collaborating components**:
  - We determine what the building blocks are
  - Put them together so they cooperate properly
  - Like building with smart Legos, some of which are pre-defined, some of which we design!



OOP Modelling: Program

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

Created (instantiated) from class definitions

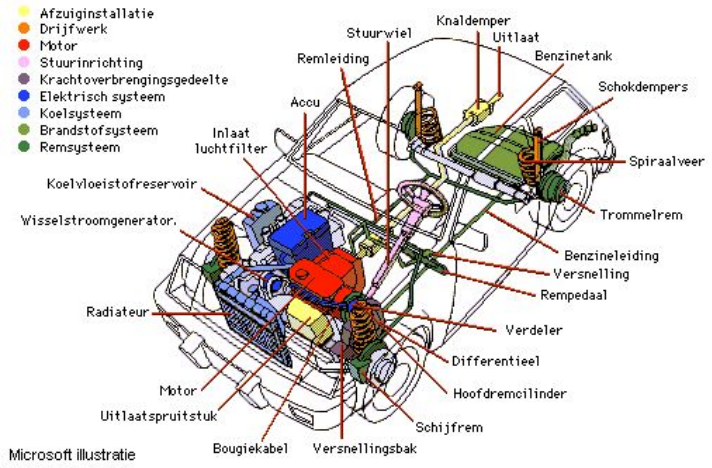


One object will send a message to another object asking it to do a particular task. The first object does not need to know how the task is done (only how to request that it be done.)

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



Abstraction

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

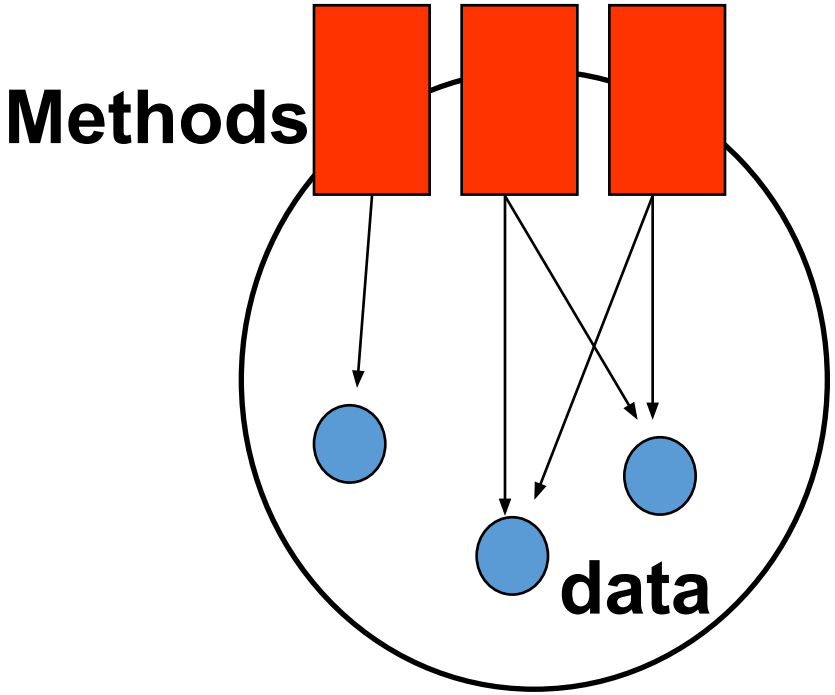
Choosing Abstraction

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

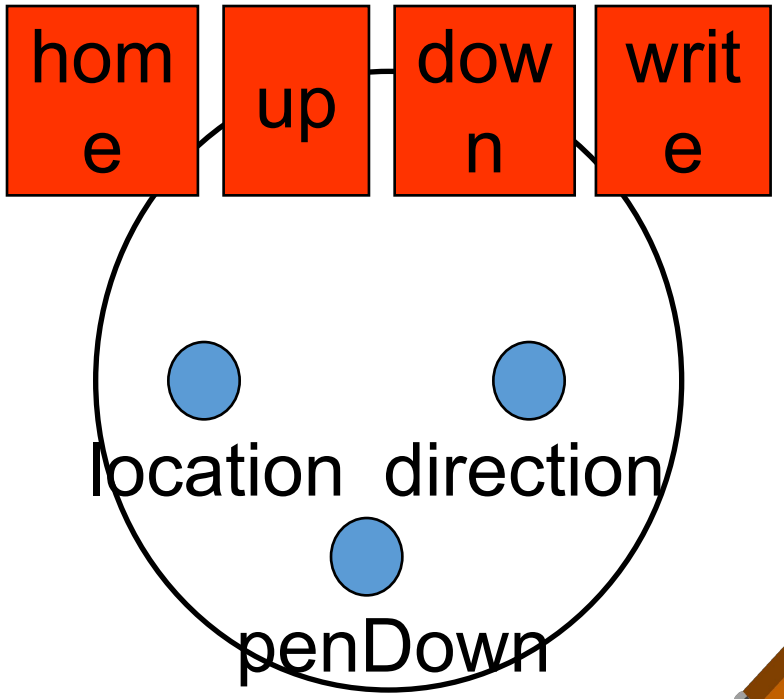
Example:

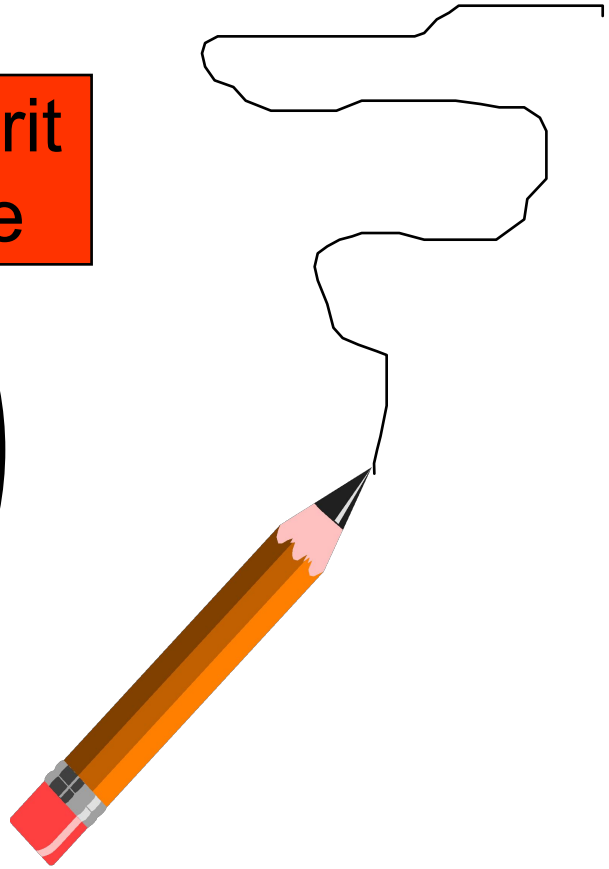
- Abstraction name: Lamp
  - Attribute: Wattage (i.e. energy usage)
  - Attribute: On/Off
- There are other possible properties (shape, color, socket size, etc.), but we have decided those are less essential
- The essential properties are determined by the problem

Example



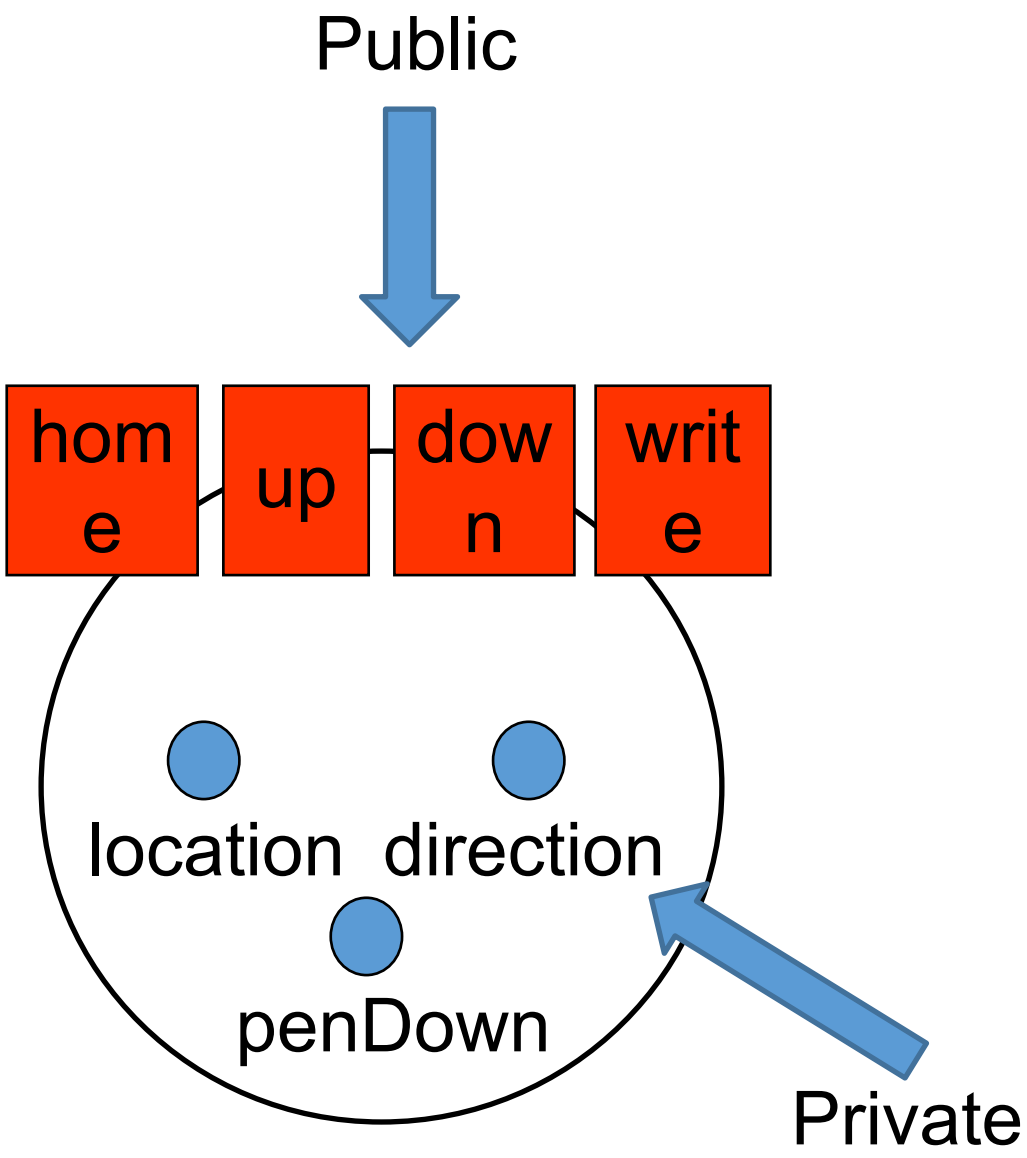
Object boundary



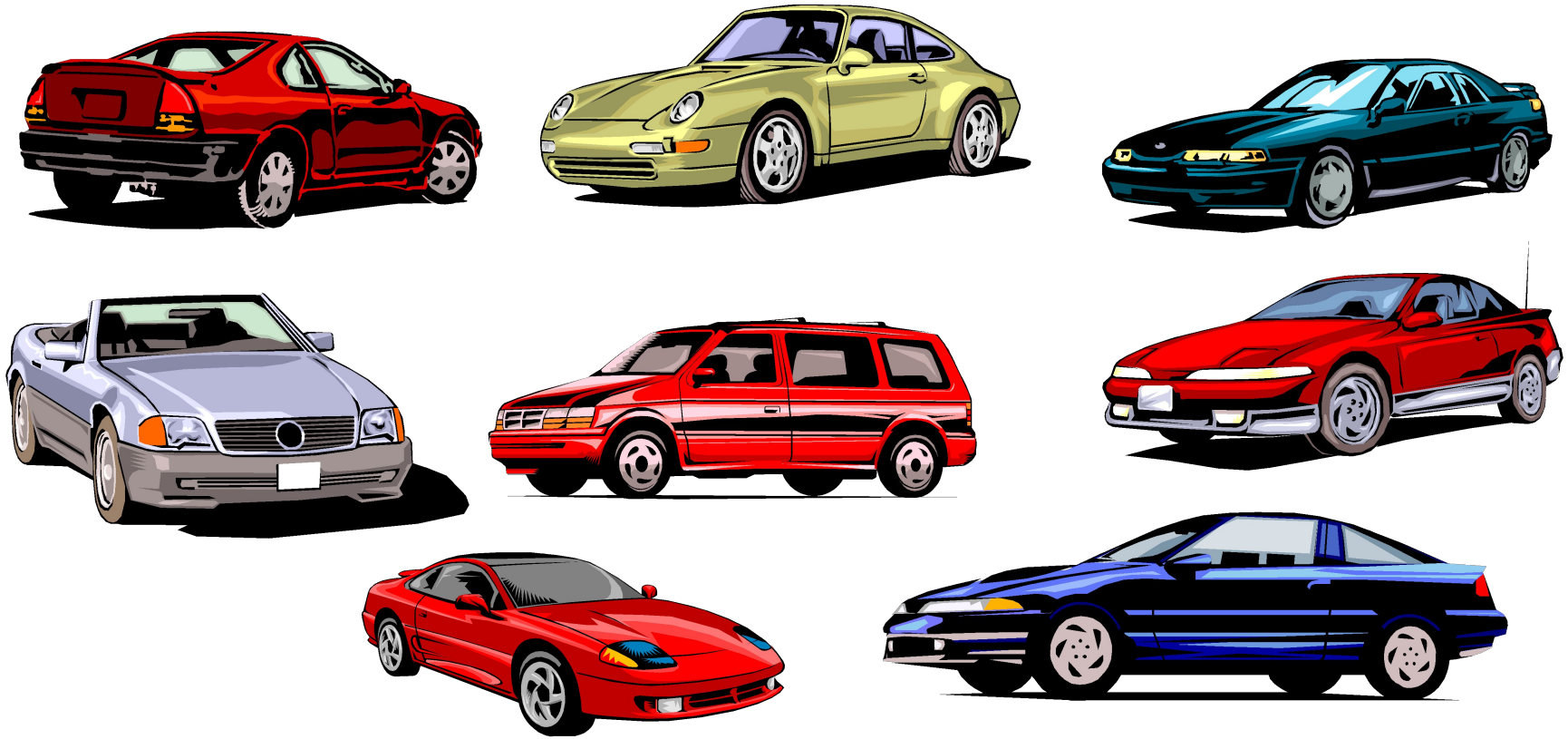


Encapsulation

- 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



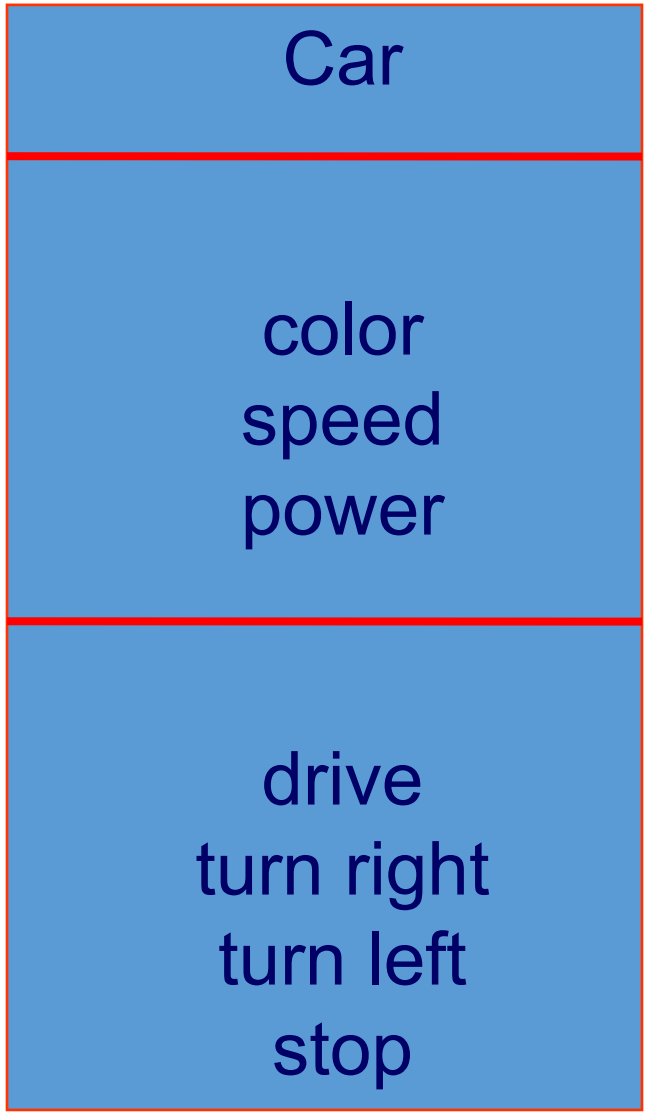
Creating Objects in Java



Defining Car Class

- What are the common attributes of cars?
- What are the common behaviors of cars?

Class Car



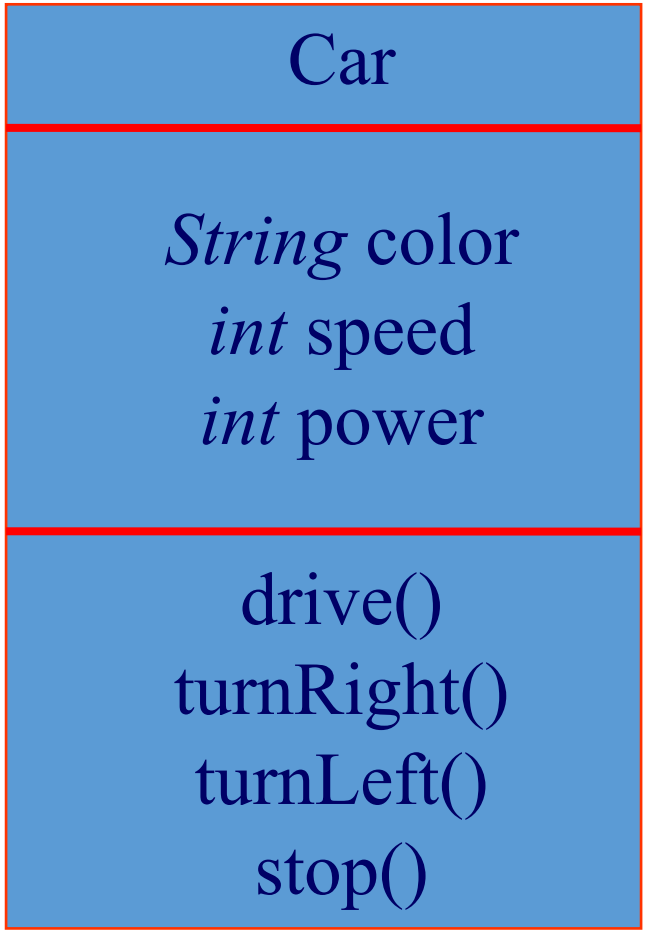
class name

attributes

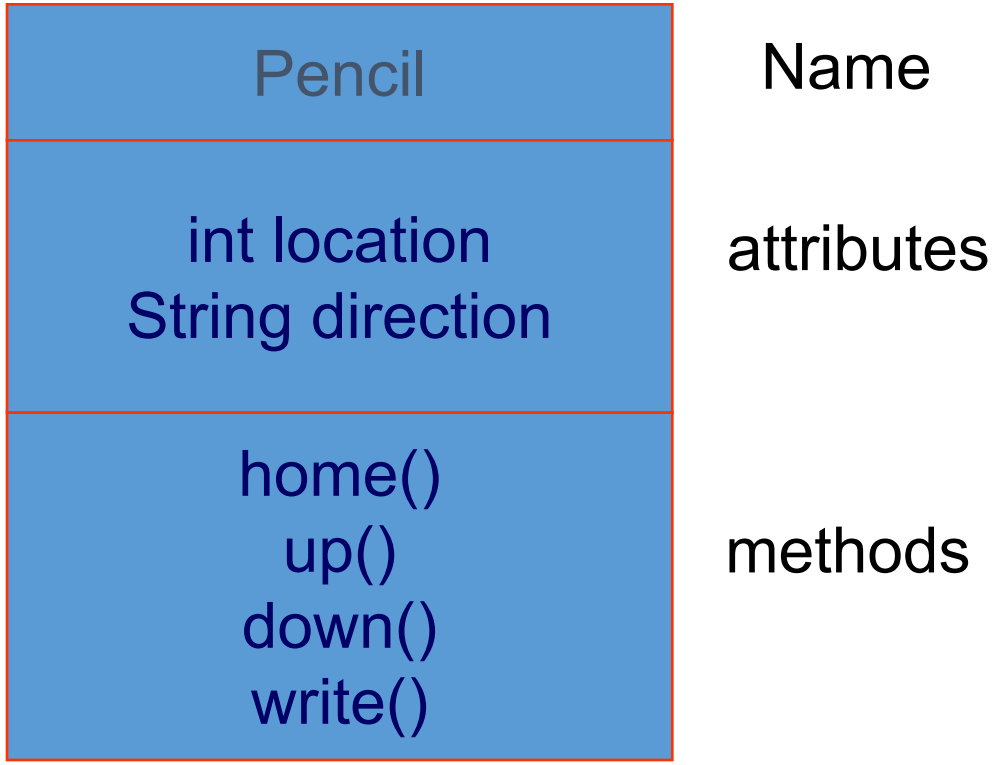
operations

Java Syntax

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

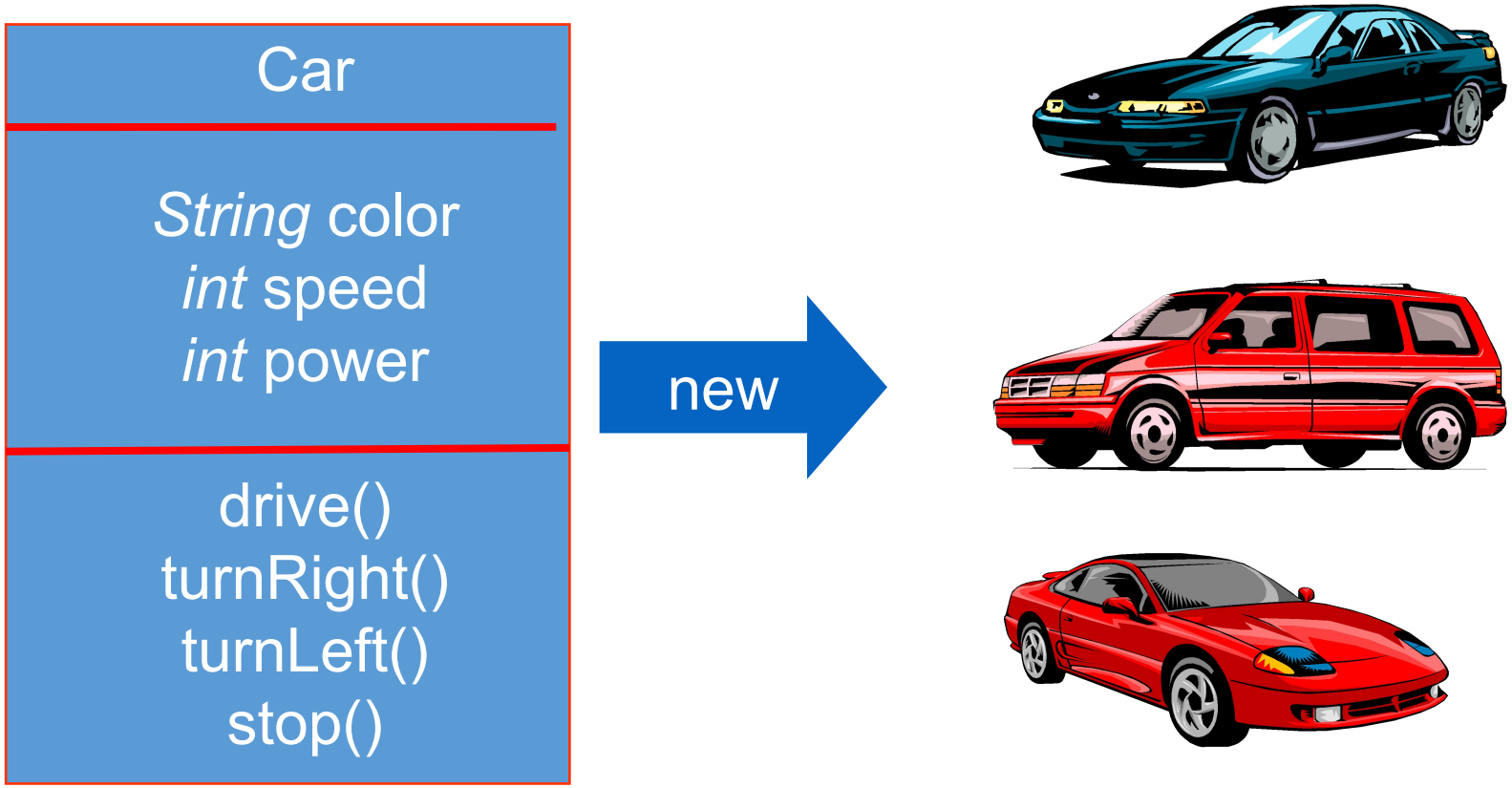


Class Pencil



Declaring objects

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

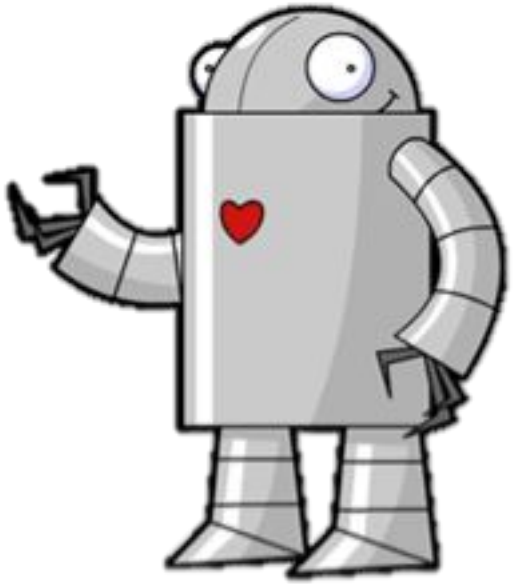


Defining and Calling Methods on Objects

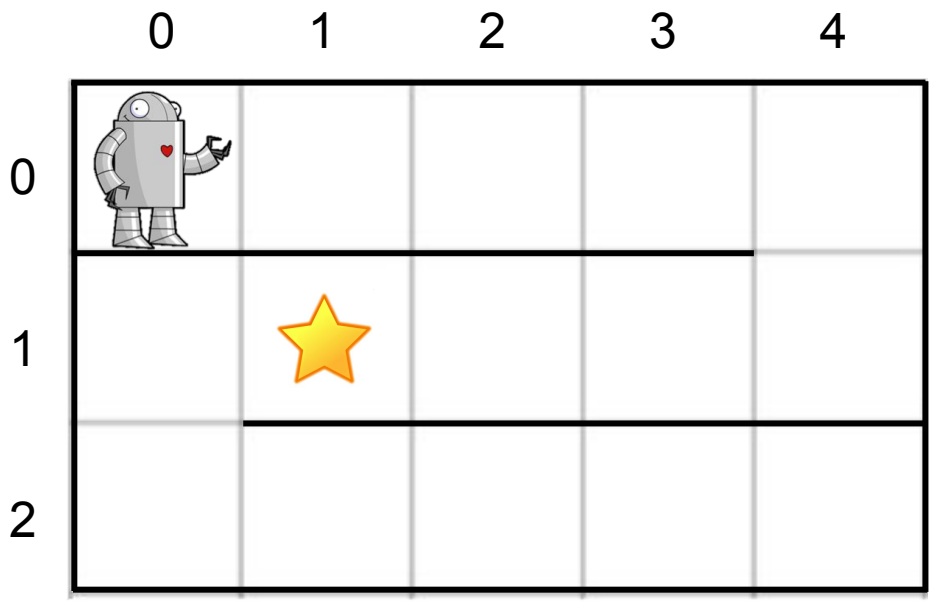
- Calling methods
- Declaring and defining a class
- Instances of a class
- Defining methods
- The `this` keyword

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 90º
- We will learn how to communicate with samBot using Java



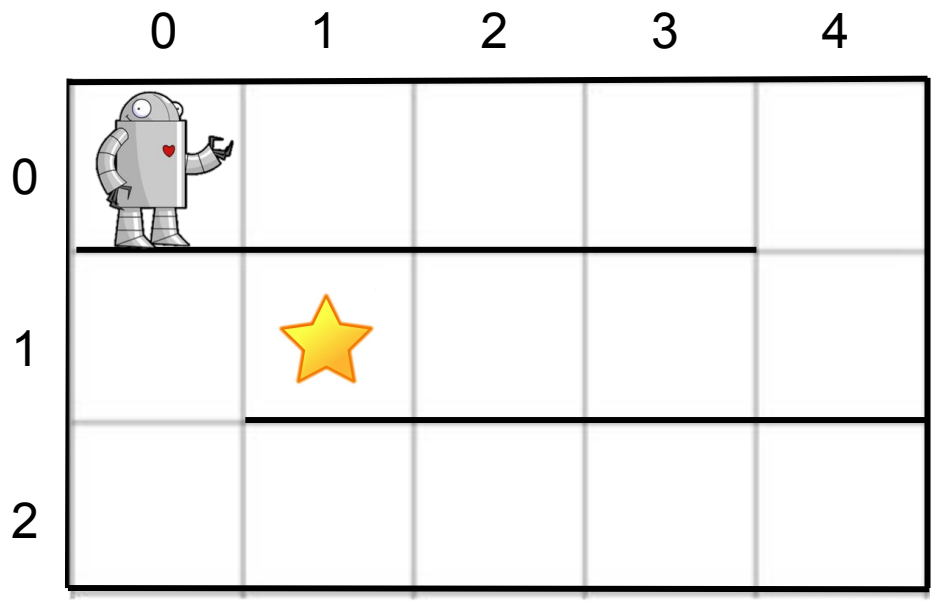
samBot’s World



- 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

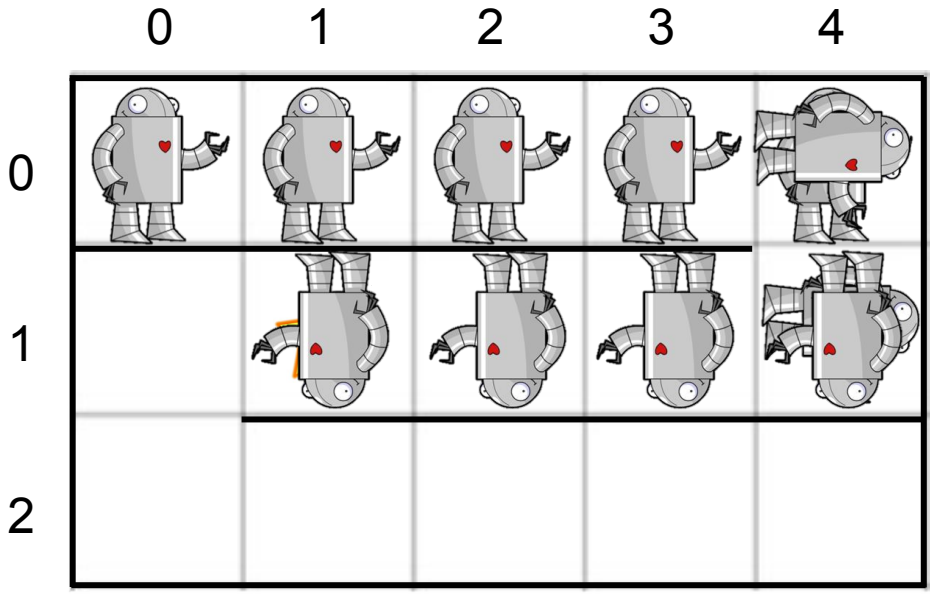
- **Goal**: move samBot from his starting position to his destination by giving him a list of instructions
- samBot only knows instructions “move forward n steps” and “turn right”
- What instructions should we give him?



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.”

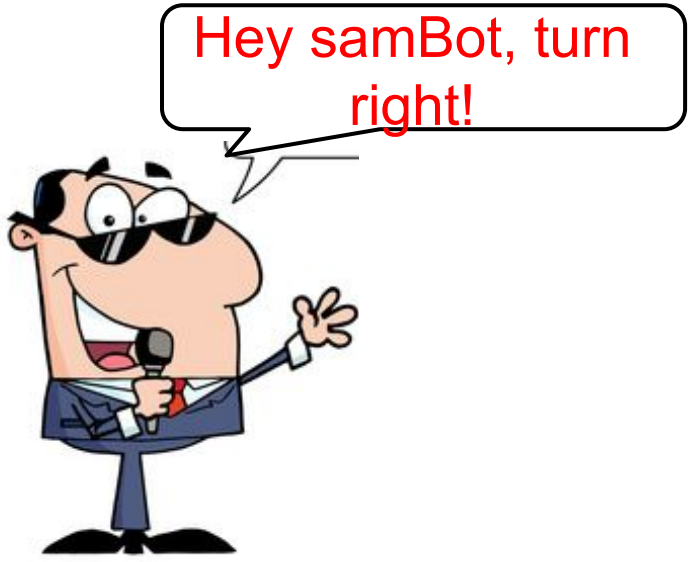


“Calling Methods”: Sending Messages in Java

- 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**

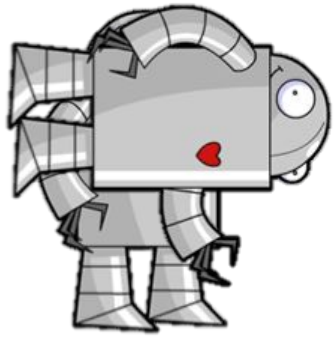
“Calling Methods”: Sending Messages in Java

- 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.**



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

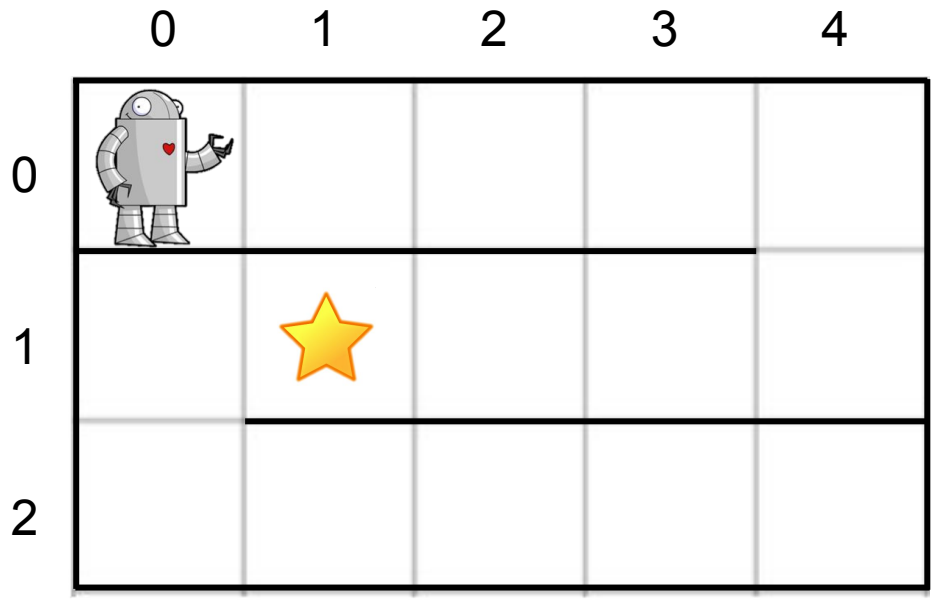
- What are those parentheses at the end of the method for?

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.

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

```
public class RobotMover {

    /* additional code elided */

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

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”

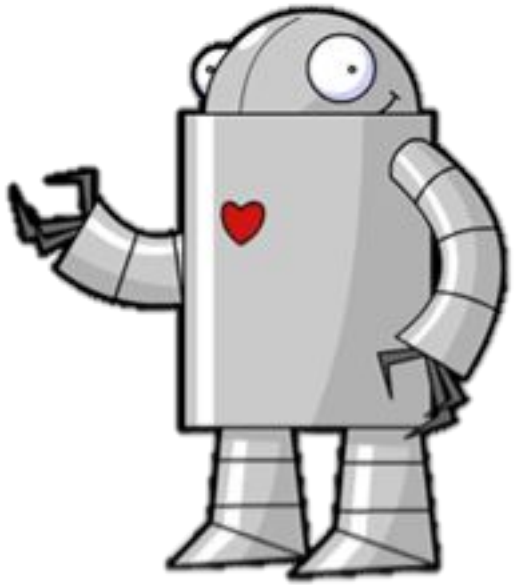
Methods of the Robot class

```
public class Robot {

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

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

     /* other code deleted */
}
```

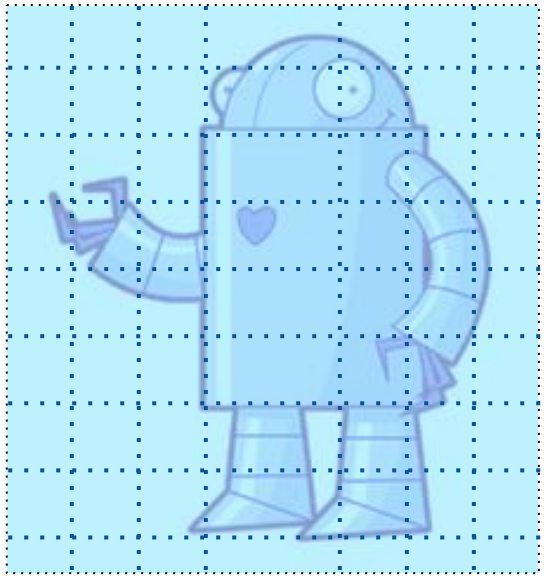
- `public void turnRight()` and `public void moveForward(int numberOfSteps)` each **declare a method**
- Since `moveForward` needs to know how many steps to move, we put `int numberOfSteps` within the parentheses
  - `int` is Java's way of saying this parameter is an "integer" (we say "of type integer")

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 `Robot`s (all **instances** of the class `Robot`) have the exact same capabilities: the methods defined in the `Robot` class



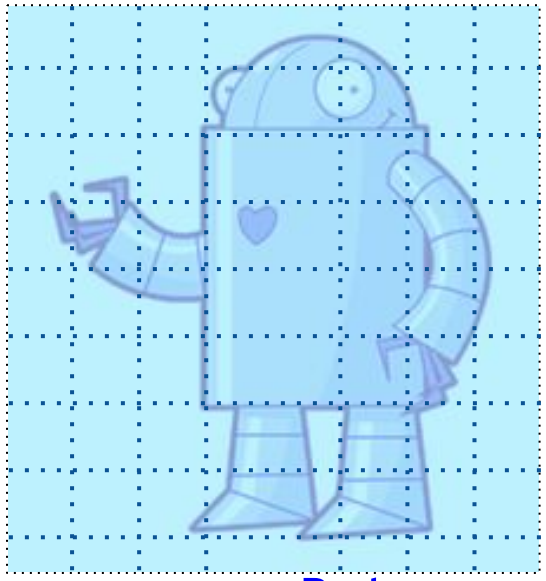
Classes and Instances (2/3)



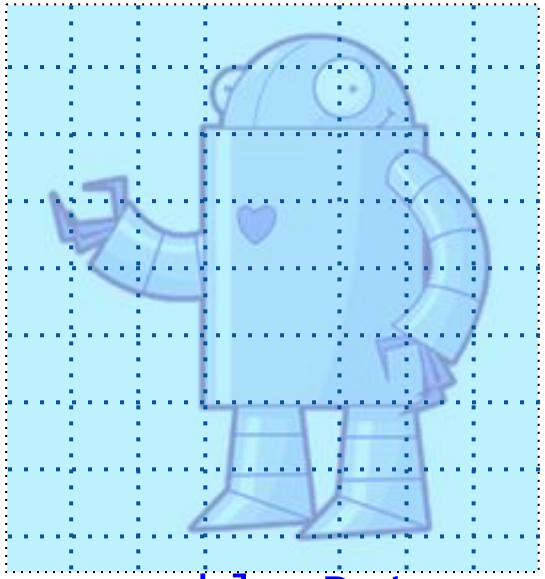
The `Robot` class is like a blueprint

Classes and Instances (3/3)

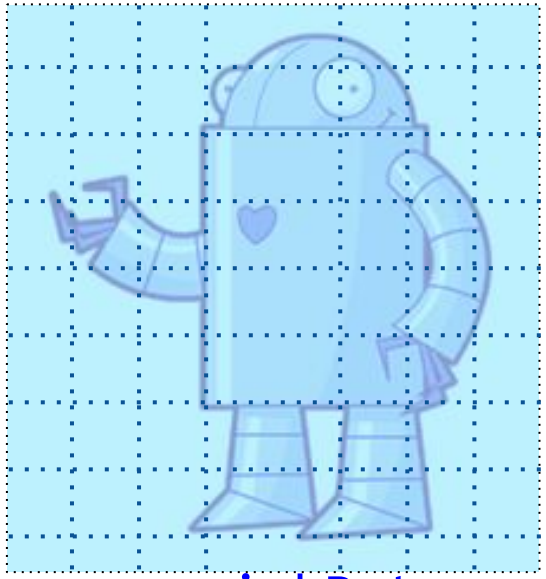
We can use the `Robot` class to build actual `Robot`s - **instances** of the class `Robot`, whose properties may vary (next lecture)



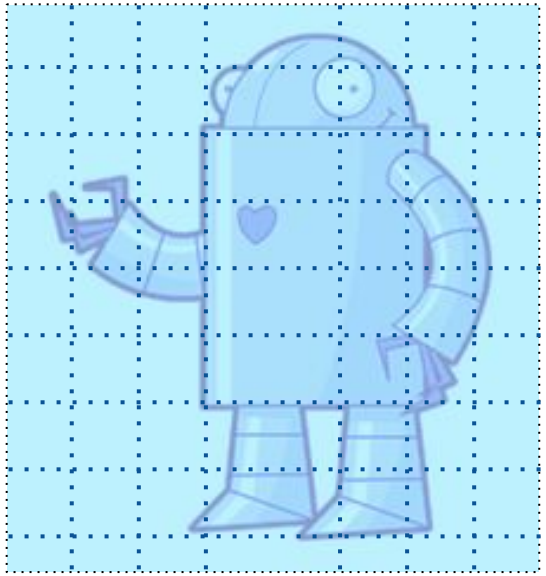
`samBot`



`blueBot`



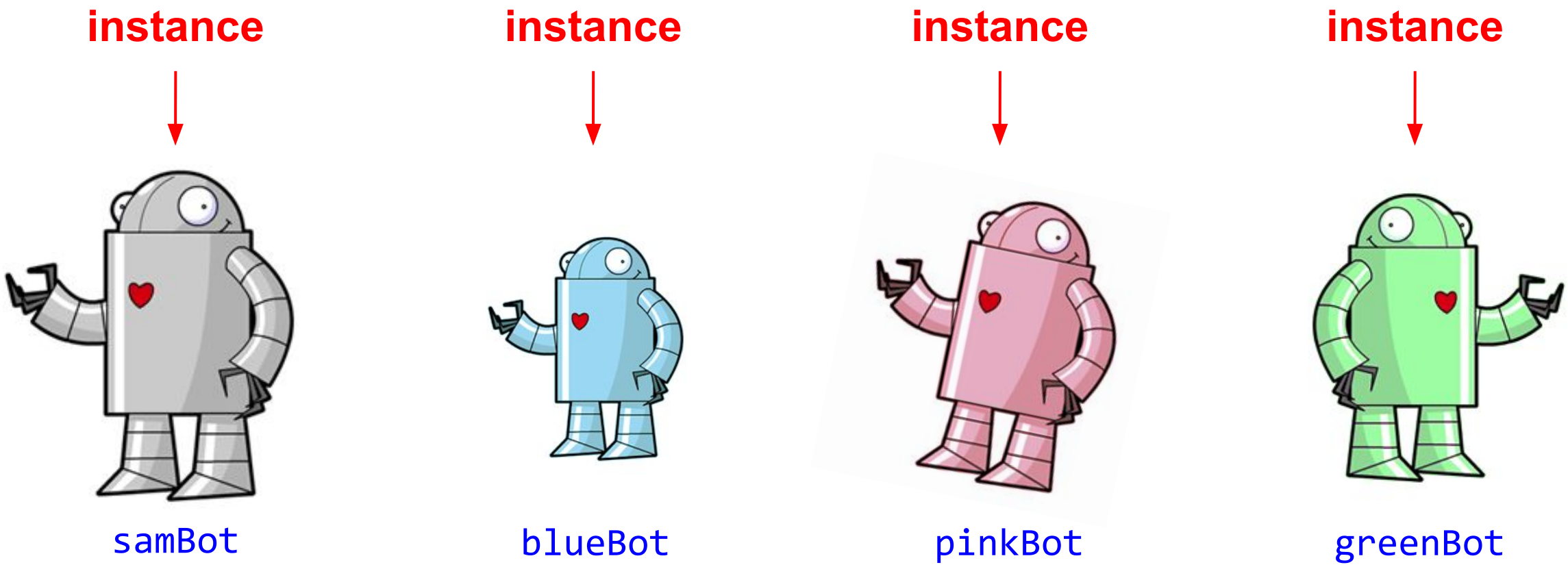
`pinkBot`



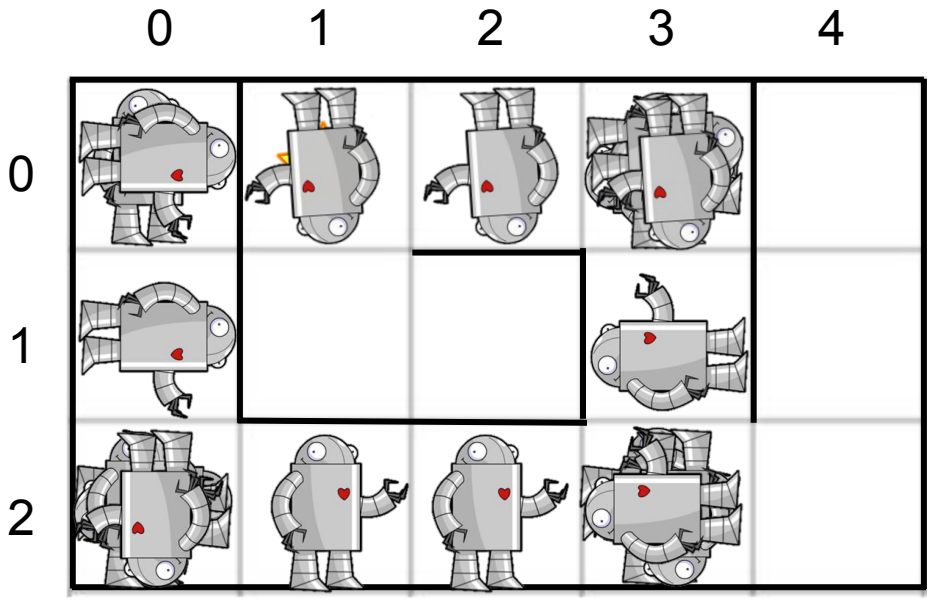
`greenBot`

Classes and Instances

Method calls are done on instances of the class



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

```
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();
    }
}
```

Class definition

Method declaration

Method definition

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 180º
- 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();
      }
}
```

Day 02: Using Classes and Objects

Slide Credits: *Internet / Chetan Arora*

Where did all these instances come from?

- We know how to send messages to an instance of a class by calling methods
- So far, we've called methods on **samBot**, an instance of `Robot`
- Where did those classes come from?
- Next: how to use a class as a blueprint to actually build instances!

Constructors (1/3)

- `Robot`s can turnRight, moveForward, etc.
- Can call any of these methods on any instance of `Robot`
- But how did these instances get created in the first place?
- Define a special kind of method in the `Robot` class: a **constructor**
- **Note: every object must have a constructor**

```
public class Robot {

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

public void moveForward(int numSteps) {
  // code that moves Robot forward
}

/* other code */
}
```

Constructors (2/3)

- A **constructor** is a special kind of method that is called whenever an object is to be “born”, i.e., created – see shortly how it is called
- Constructor’s name is always the same as name of class
- If the class is called “Robot”, its constructor needs to be called “Robot”. If the class is called “Dog”, its constructor had better be called “Dog”

```
public class Robot {

 public Robot () {
    // this is the constructor!
 }

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

 public void moveForward(int numSteps) {
   // code that moves Robot forward
 }

 /* other code */
}
```

Constructors (3/3)

- Constructors are special methods: used only once, to create the instance
- And we never specify a return value in its declaration
- Constructor for `Robot` does not take in any parameters (notice empty parentheses)
- Constructors can, and often do, take in parameters– later…

```
public class Robot {

 public Robot () {
    // this is the constructor!
 }

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

 public void moveForward(int numSteps) {
   // code that moves Robot forward
 }

 /* other code */
}
```

Instantiating Objects (1/3)

- Now that the `Robot` class has a constructor, we can create instances of `Robot` !
- Here's how we create a `Robot` in Java:

```
new Robot();
```

- This means "use the `Robot` class as a blueprint to create a new `Robot` instance"
- `Robot()` is a call to `Robot`'s constructor, so any code in the constructor will be executed as soon as you create a `Robot`

Instantiating Objects (2/3)

- We refer to “creating” an object as **instantiating** it
- When we say:

```
new Robot();
```

- … We’re **creating an instance** of the `Robot` class, a.k.a. **instantiating** a new `Robot`
- Where exactly does this code get executed?

Invoking Methods

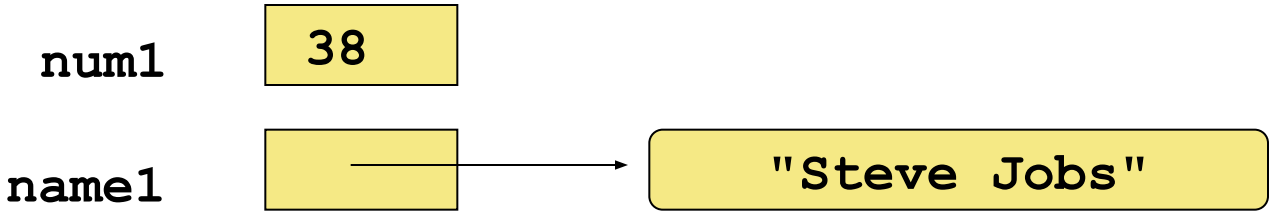
- We've seen that once an object has been instantiated, we can use the *dot operator* to invoke its methods

```
count = title.length()
```

- A method may *return a value*, which can be used in an assignment or expression
- A method invocation can be thought of as asking an object to perform a service

References

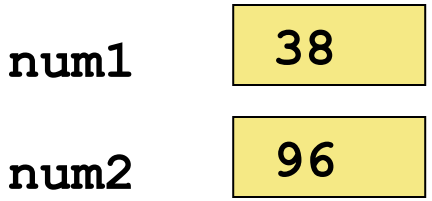
- Note that a primitive variable contains the value itself, but an object variable contains the address of the object
- An object reference can be thought of as a pointer to the location of the object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically



Assignment Revisited

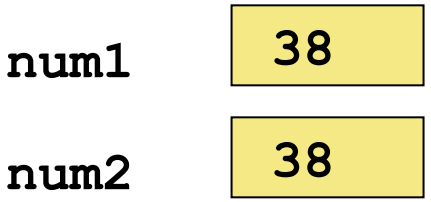
- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:

Before:




```
num2 = num1;
```

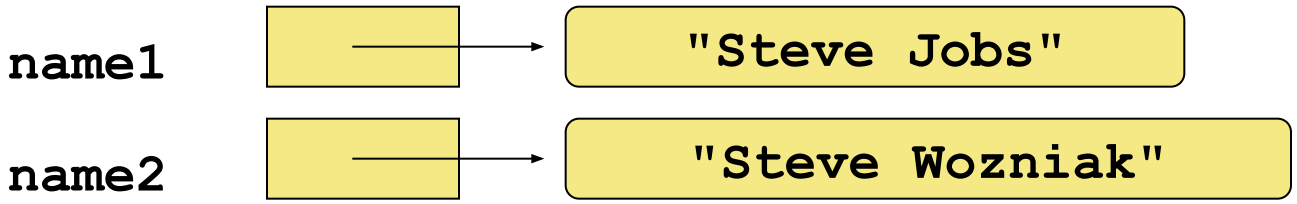
After:



Reference Assignment

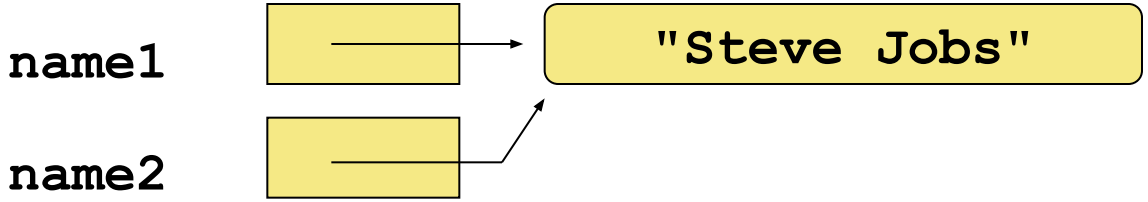
- For object references, assignment copies the address:

Before:




```
name2 = name1;
```

After:



Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- That creates an interesting situation: one object can be accessed using multiple reference variables
- Aliases can be useful, but should be managed carefully
- Changing an object through one reference changes it for all of its aliases, because there is really only one object

Garbage Collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- The object is useless, and therefore is called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In other languages, the programmer is responsible for performing garbage collection

Class Libraries

- A *class library* is a collection of classes that we can use when developing programs
- The *Java standard class library* is part of any Java development environment
- Its classes are not part of the Java language per se, but we rely on them heavily
- Various classes we've already used (`System`, `String`) are part of the Java standard class library
- Other class libraries can be obtained through third party vendors, or you can create them yourself

Packages

- The classes of the Java standard class library are organized into *packages*
- Some of the packages in the standard class library are:

Package	Purpose
java.lang	**General support**
java.applet	**Creating applets for the web**
java.awt	**Graphics and graphical user interfaces**
javax.swing	**Additional graphics capabilities**
java.net	**Network communication**
java.util	**Utilities**
javax.xml.parsers	**XML document processing**

The import Declaration

- When you want to use a class from a package, you could use its *fully qualified name*

```
java.util.Scanner;

java.util.Random;
```

Or you can *import* the class, and then use just the class name

```
import java.util.Scanner;
```

- To import all classes in a particular package, you can use the * wildcard character

```
import java.util.*;
```

The import Declaration

- All classes of the `java.lang` package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the `System` or `String` classes explicitly in earlier programs
- The `Scanner` class, on the other hand, is part of the `java.util` package, and therefore must be imported

Class based Access Privileges: Access Modifiers

- Determine whether other classes can use a particular field or invoke a particular method
- The first data column indicates whether the class itself has access to the member defined by the access level.
- The second column indicates whether classes in the same package as the class have access to the member.
- The third column indicates whether subclasses of the class declared outside this package have access to the member.
- The fourth column indicates whether all classes have access to the member.

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`no modifier`	Y	Y	N	N
`private`	Y	N	N	N

The Random Class

- The `Random` class is part of the `java.util` package
- It provides methods that generate pseudorandom numbers
- A `Random` object performs complicated calculations based on a *seed value* to produce a stream of seemingly random values

```
    Random generator = new Random();
int num1;
float num2;

num1 = generator.nextInt();
System.out.println ("A random integer: " + num1);

num1 = generator.nextInt(10);
System.out.println ("From 0 to 9: " + num1);

num1 = generator.nextInt(15) + 20;
System.out.println ("From 20 to 34: " + num1);

num1 = generator.nextInt(20) - 10;
System.out.println ("From -10 to 9: " + num1);

num2 = generator.nextFloat();
System.out.println ("A random float [between 0-1]: " + num2);

num2 = generator.nextFloat() * 6;  // 0.0 to 5.999999
num1 = (int) num2 + 1;
System.out.println ("From 1 to 6: " + num1);
```

Interactive Programs

- Programs generally need input on which to operate
- The `Scanner` class provides convenient methods for reading input values of various types
- A `Scanner` object can be set up to read input from various sources, including the user typing values on the keyboard
- Keyboard input is represented by the `System.in` object

Reading Input

- The following line creates a Scanner object that reads from the keyboard:

```
Scanner scan = new Scanner (System.in);
```

- The `new` operator creates the `Scanner` object
- Once created, the `Scanner` object can be used to invoke various input methods, such as:

```
answer = scan.nextLine();
```

Reading Input

- The `Scanner` class is part of the `java.util` class library, and must be imported into a program to be used
- The `nextLine` method reads all of the input until the end of the line is found

Echo.java

```
import java.util.Scanner;
public class Echo {
    //  Reads a character string from the user and prints it.
    public static void main (String[] args) {
     String message;
     Scanner scan = new Scanner (System.in);

     System.out.println ("Enter a line of text:");
     message = scan.nextLine();

     System.out.println ("You entered: \"" + message + "\"");
  }
}
```

Input Tokens

- Unless specified otherwise, *white space* is used to separate the elements (called *tokens*) of the input
- White space includes space characters, tabs, new line characters
- The `next` method of the `Scanner` class reads the next input token and returns it as a string
- Methods such as `nextInt` and `nextDouble` read data of particular types

```
public static void main (String[] args)   {
    int miles;
    double gallons, mpg;

    Scanner scan = new Scanner (System.in);

    System.out.print ("Enter the number of miles: ");
    miles = scan.nextInt();

    System.out.print ("Enter the gallons of fuel used: ");
    gallons = scan.nextDouble();

    mpg = miles / gallons;
```

Static Class Members

- A static method can be invoked through its class name
- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25)
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the `static` modifier
- It associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called *class methods* and static variables are sometimes called *class variables*
- Let's carefully consider the implications of each

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

```
private static float price;
```

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

Static Methods

```
class Helper
{
   public static int cube (int num)
   {
      return num * num * num;
   }
}
```

Because it is declared as static, the method can be invoked as

```
value = Helper.cube(5);
```

Static Class Members

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

Static Class Members

- Static methods and static variables often work together
- The following example keeps track of how many objects have been created using a static variable, and makes that information available using a static method

```
class MyClass {
   private static int count = 0;

   public MyClass () {
      count++;
   }
   public static int getCount () {
      return count;
   }
}
```

```
      MyClass obj;

      for (int scan=1; scan <= 10; scan++)
         obj = new MyClass();

      System.out.println ("Objects created: " +
  MyClass.getCount());
```

Student Id prolem

- Let’s suppose we have a Student class
- How do we assign unique student id’s to each student object that we create?
- What if we also want to get the latest Student created? Like:

```
public static String getLatestStudent()
```

The this Reference

- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used in a method called `tryMe`, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

- **In the first invocation, the `this` reference refers to `obj1`; in the second it refers to `obj2`**

The this reference

- The `this` reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names

- The constructor of the `Account` class (from Chapter 4) could have been written as follows:

```
public Account (Sring name, long acctNumber,
                double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

References

- Recall that an object reference holds the memory address of an object

- Rather than dealing with arbitrary addresses, we often depict a reference graphically as a “pointer” to an object

```
ChessPiece bishop1 = new ChessPiece();
```

References

- Things you can do with a reference:
    - Declare it : String st;
    - Assign a new value to it
        - st = new String(“java”);
        - st = st2;
        - st = null;
    - Interact with the object using “dot” operator : st.length()
    - Check for equivalence
        - (st == st2)
        - (st == null)

The null Reference

- An object reference variable that does not currently point to an object is called a *null reference*
- The reserved word `null` can be used to explicitly set a null reference:

```
name = null;
```

or to check to see if a reference is currently null:

```
if (name == null)
    System.out.println ("Invalid");
```

The null Reference

- An object reference variable declared at the class level (an instance variable) is automatically initialized to null
- The programmer must carefully ensure that an object reference variable refers to a valid object before it is used
- Attempting to follow a null reference causes a `NullPointerException` to be thrown
- Usually a compiler will check to see if a local variable is being used without being initialized

Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the argument (the value passed) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement
- When an object is passed to a method, the argument and the formal parameter become aliases of each other

Passing Objects to Methods

- What a method does with a parameter may or may not have a permanent effect (outside the method)
- Note the difference between changing the internal state of an object versus changing which object a reference points to

ParameterPassing

```
ParameterTester tester = new ParameterTester();

    int a1 = 111;
    Num a2 = new Num (222);
    Num a3 = new Num (333);

    System.out.println ("Before calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

    tester.changeValues (a1, a2, a3);

    System.out.println ("After calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
```

ParameterTester

```
class ParameterTester
{
   //-----------------------------------------------------------------
   //  Modifies the parameters, printing their values before and
   //  after making the changes.
   //-----------------------------------------------------------------
   public void changeValues (int f1, Num f2, Num f3)
   {
      System.out.println ("Before changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

      f1 = 999;
      f2.setValue(888);
      f3 = new Num (777);
```

Num

```
class Num {
   private int value;

public Num (int update) {
      value = update;
   }

public void setValue (int update)
   {
      value = update;
   }

public String toString () {
      return value + "";
   }
}
```

Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters

Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
    return x + .375;
}

float tryMe(int x, float y)
{
    return x*y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```

Method Overloading

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```

Overloading Methods

- The return type of the method is <u>not</u> part of the signature
- That is, overloaded methods cannot differ only by their return type
- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

Overloading Methods

- Constructors can be overloaded
- An overloaded constructor provides multiple ways to set up a new object

Day 03: GUI Programming

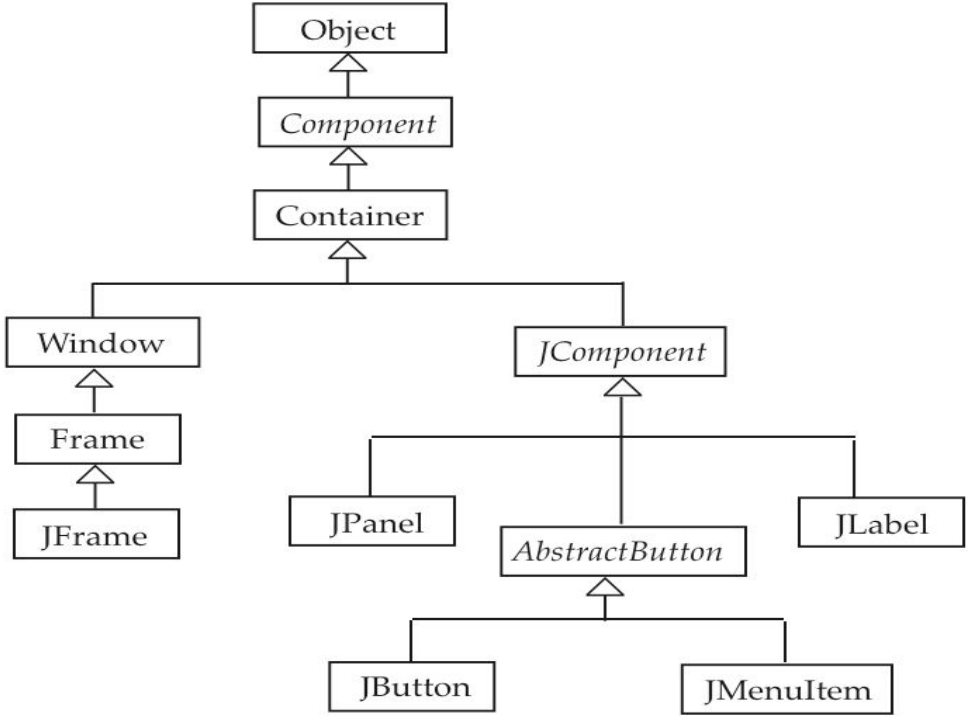
Slide Credits: *Internet / Chetan Arora*

Advanced Programming. All slides copyright: Chetan Arora

Introduction

- Java 1.0: Contained AWT (Abstract Window Toolkit) class library for basic GUI programming.
- SWING: GUI Toolkit available from Java 1.1
- Swing has a rich and convenient set of user interface elements.
- Swing has few dependencies on the underlying platform; it is therefore less prone to platform-specific bugs.
- Swing gives a consistent user experience across platforms.

Inheritance Hierarchy for Frame and Component Classes



Create a Frame

- Top-level Window
- JFrame Class in Swing (extends Frame Class)
- Not painted on a canvas

```
public class SizedFrameTest
{
    public static void main(String[] args)
    {

                JFrame frame = new JFrame();
                frame.setTitle("BasicFrame");
                frame.setSize(300,400);
                frame.setIconImage(new
ImageIcon("smiley.jpg").getImage());
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);

     }
  }
```

To determine an appropriate size for a frame, first find out the screen size. Call the static `getDefaultToolkit` method of the `Toolkit` class to get the `Toolkit` object. (The `Toolkit` class is a dumping ground for a variety of methods interfacing with the native windowing system.) Then call the `getScreenSize` method, which returns the screen size as a `Dimension` object. A `Dimension` object simultaneously stores a width and a height, in public (!) instance variables `width` and `height`. Then you can use a suitable percentage of the screen size to size the frame. Here is the code:

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
setSize(screenWidth / 2, screenHeight / 2);
```

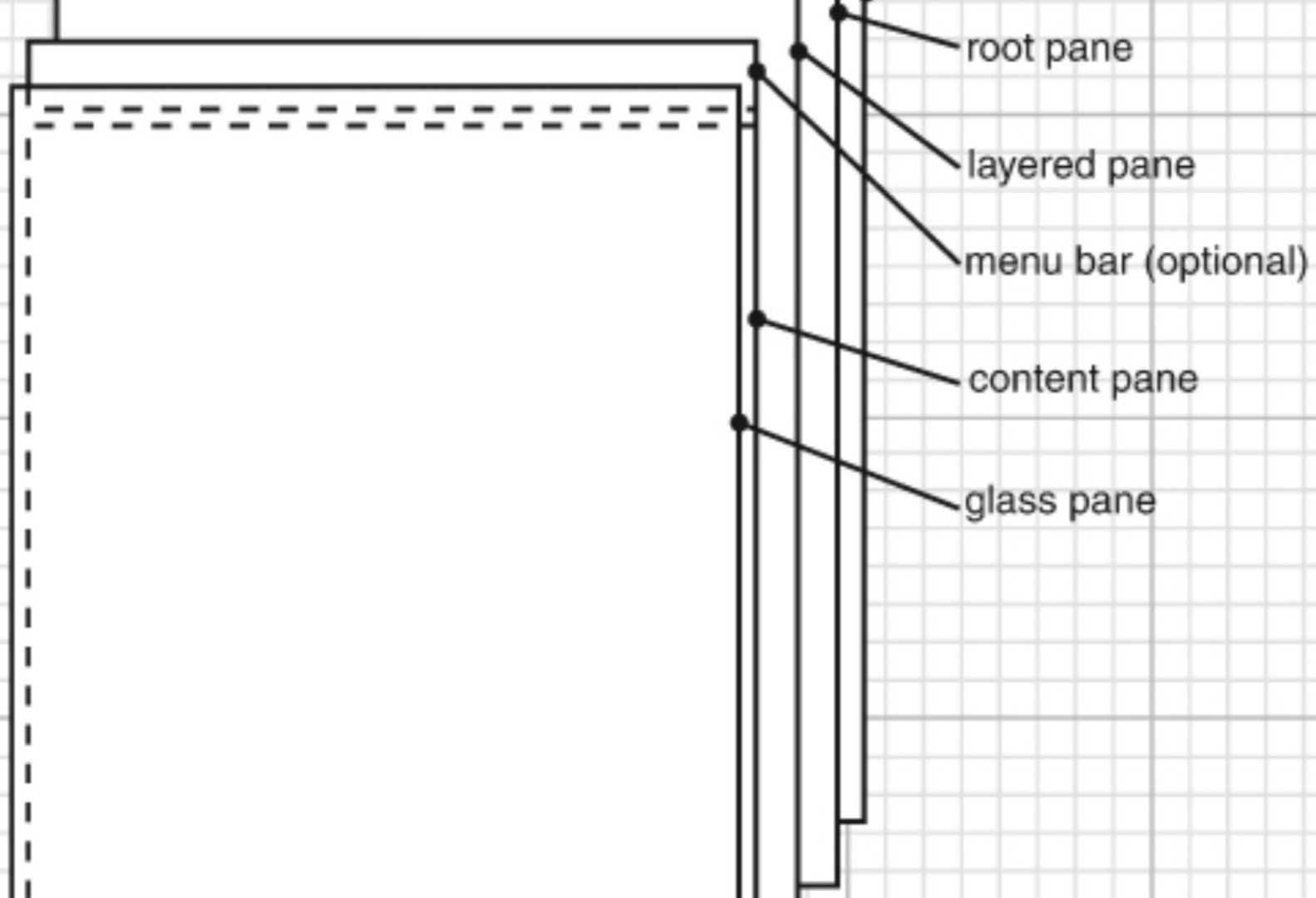
Display Information in Component

```
}
public class SimpleFrameTest
{
  public static void main(String[] args)
  {
     EventQueue.invokeLater(() ->
       {
          var frame = new SimpleFrame();
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          frame.setVisible(true);
       });
  }
}
```

Display Information in Component

- Could draw the message string directly onto a frame
  - Bad programming practice
  - Frames are really designed to be containers for components
  - Normally draw on another component added to the frame
  - When designing a frame, you add components into the content pane

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```


root pane
layered pane
menu bar (optional)
content pane
glass pane

Display Information in Component

```
import javax.swing.*;
import java.awt.*;

public class NotHelloWorld
{
    public static void main(String[] args)
    {
       EventQueue.invokeLater(new Runnable()
          {
             public void run()
             {
                JFrame frame = new NotHelloWorldFrame();
                frame.setTitle("NotHelloWorld");
             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
             }
          });
     }
  }
```

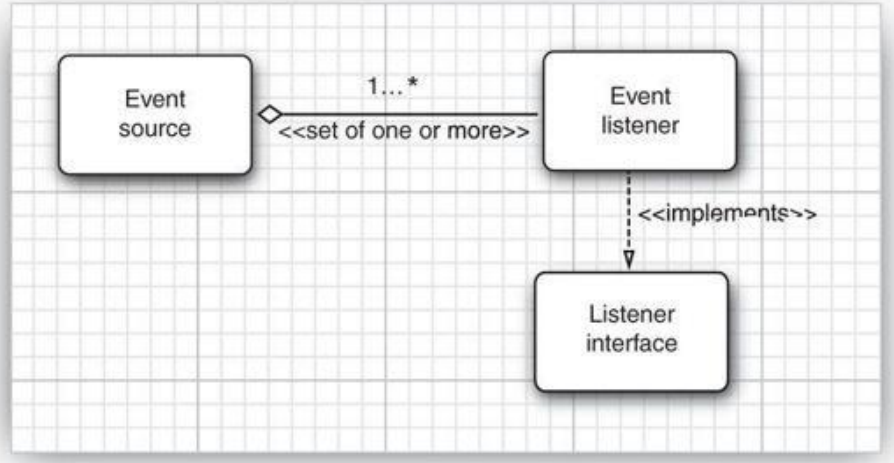
Display Information in Component

```
/**
      * A frame that contains a message
     panel
      */
     class NotHelloWorldFrame extends
     JFrame
     {
        public NotHelloWorldFrame()
        {
           add(new
     NotHelloWorldComponent());
           pack();
        }
     }
```

```
/**
* A component that displays a message.
*/
class NotHelloWorldComponent extends JComponent
  {
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT =200;
    public void paintComponent(Graphics g)
     {
      g.drawString("Not a Hello, World program",
     MESSAGE_X, MESSAGE_Y);
     }
     public Dimension getPreferredSize() {
     return new Dimension(DEFAULT_WIDTH,
      DEFAULT_HEIGHT); }
    }
```

Event Handling

Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks



Event Handling

- A listener object is an instance of a class that implements a special interface called (naturally enough) a listener interface.
- An event source is an object that can register listener objects and send them event objects.
- The event source sends out event objects to all registered listeners when that event occurs.
- The listener objects will then use the information in the event object to determine their reaction to the event

Event Handling

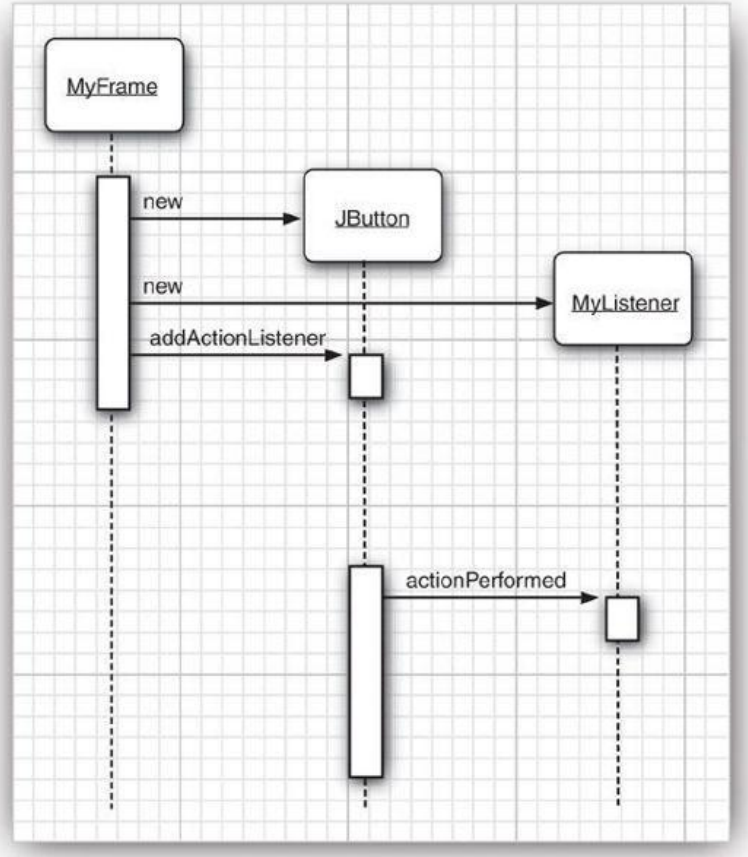
- Specify Listener

```
ActionListener listener = . . .;
JButton button = new JButton("Ok");
button.addActionListener(listener);
```

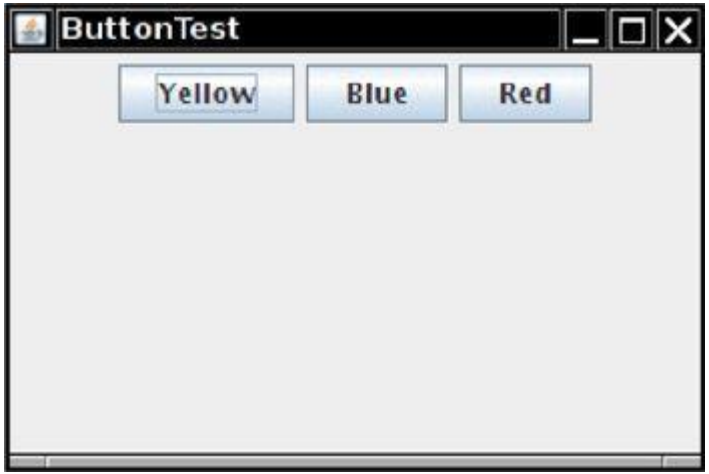
Event Handling

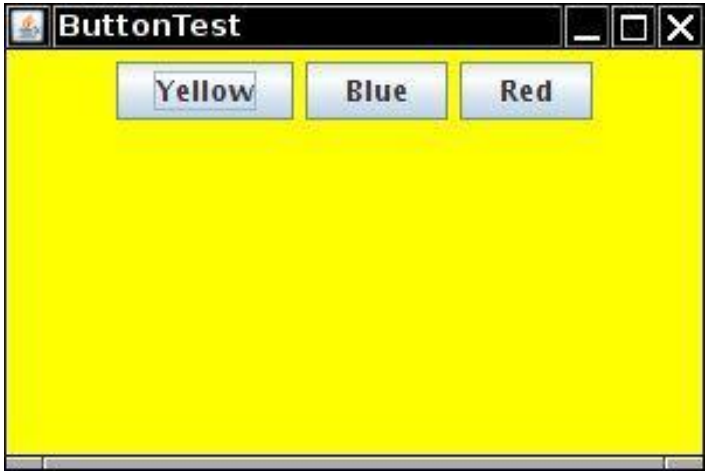
- To implement the ActionListener interface, the listener class must have a method called actionPerformed that receives an ActionEvent object as a parameter.

```
class MyListener implements ActionListener
{
   . . .
   public void actionPerformed(ActionEvent event)
   {
      // reaction to button click goes here
      . . .
   }
}
```


MyFrame
new
JButton
new
MyListener
addActionListener
actionPerformed

Example: Handling a Button Click





```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
 /**
  * A frame with a button panel
  */
 public class ButtonFrame extends JFrame
  {
    private JPanel buttonPanel;
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

     public ButtonFrame()
     {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        JButton yellowButton = new
JButton("Yellow");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        buttonPanel = new JPanel();
```

```
// add buttons to panel
buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);

// add panel to frame
add(buttonPanel);

// create button actions
ColorAction yellowAction = new
ColorAction(Color.YELLOW);

ColorAction blueAction = new
ColorAction(Color.BLUE);

ColorAction redAction = new
ColorAction(Color.RED);

// associate actions with buttons
yellowButton.addActionListener(yellowActi
on);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
}
```

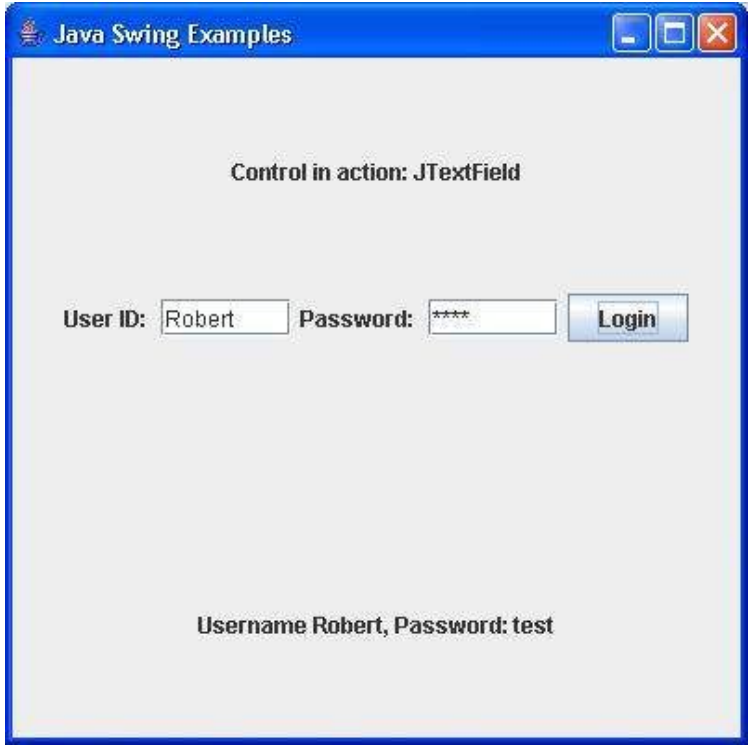
```
   /**
    * An action listener that sets the panel's background color.
    */
   private class ColorAction implements ActionListener
   {
      private Color backgroundColor;

      public ColorAction(Color c)
      {
         backgroundColor = c;
      }

      public void actionPerformed(ActionEvent event)
      {
         buttonPanel.setBackground(backgroundColor);
      }
   }
}
```

Alternative

```
public void actionPerformed(ActionEvent e) {

        Color color = display.getBackground();

        int red = color.getRed();
        int green = color.getGreen();
        int blue = color.getBlue();

        if (e.getActionCommand().equals("Red")) {
            if (red == 0) {
                red = 255;
            } else {
                red = 0;
            }
        }
```

```
if (e.getActionCommand().equals("Green")) {
            if (green == 0) {
                green = 255;
            } else {
                green = 0;
            }
        }

if (e.getActionCommand().equals("Blue")) {
            if (blue == 0) {
                blue = 255;
            } else {
                blue = 0;
            }
        }
Color setCol = new Color(red, green, blue);
        display.setBackground(setCol);
    }
```

Using TextFields: JTextFields Example



Create Basic GUI

```
private void prepareGUI(){
     mainFrame = new JFrame("Java Swing Examples");
     mainFrame.setSize(400,400);
     mainFrame.setLayout(new GridLayout(3, 1));
     mainFrame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent windowEvent){
           System.exit(0);
        }
       });
```

Create Basic GUI: Add components on the Frame

Continued...

```
headerLabel = new JLabel("", JLabel.CENTER);
statusLabel = new JLabel("",JLabel.CENTER);

statusLabel.setSize(350,100);

controlPanel= new JPanel();

controlPanel.setLayout(new FlowLayout());

mainFrame.add(headerLabel);

mainFrame.add(controlPanel);

mainFrame.add(statusLabel);

mainFrame.setVisible(true);
```

Adding TextFields: JTextFields

```
private void showTextFieldDemo(){
     headerLabel.setText("Control in action: JTextField");
     JLabel namelabel= new JLabel("User ID: ", JLabel.RIGHT);
     JLabel passwordLabel = new JLabel("Password:", JLabel.CENTER);
     final JTextField userText = new JTextField(6);
     final JPasswordField passwordText = new JPasswordField(6);
```

Adding Button and Event handling

```
JButton loginButton = new JButton("Login");
loginButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      String data = "Username " + userText.getText();
      data += ", Password: " + new String(passwordText.getPassword());
      statusLabel.setText(data);
   }
});
controlPanel.add(namelabel);
controlPanel.add(userText);
controlPanel.add(passwordLabel);
controlPanel.add(passwordText);
controlPanel.add(loginButton);
mainFrame.setVisible(true);
}
```

Abstract class, interface, Extends and implements keywords, abstract keyword.
————————————————-

In Java, the `implements` and `extends` keywords are used to implement interfaces and extend classes, respectively. Both keywords are used in class declarations to indicate the relationship between classes and interfaces.

1. `implements`:
 - When a class implements an interface, it is agreeing to adhere to the contract defined by the interface. It means the class will provide concrete implementations for all the abstract methods declared in the interface.
 - A class can implement multiple interfaces by separating them with commas. - An interface provides a way to define a common set of methods that unrelated classes can implement, enabling polymorphism and achieving loose coupling between classes.

Example of implementing an interface:

```java
interface Drawable {
 void draw();
}

class Circle implements Drawable {
 @Override
 public void draw() {
 System.out.println("Drawing a circle");
 }
}

class Square implements Drawable {
 @Override
 public void draw() {
 System.out.println("Drawing a square");
 }
}
```

2. `extends`:
 - When a class extends another class, it inherits all the fields and methods (except private ones) from the parent class, allowing the subclass to reuse and extend the behavior of the superclass. - Java supports single inheritance, meaning a class can only extend one superclass. However, it can implement multiple interfaces (using `implements`) to achieve multiple inheritance of behavior.

Example of extending a class:

```java
class Vehicle {
 void start() {
 System.out.println("Vehicle is starting...");
 }
}

class Car extends Vehicle {
 void drive() {
 System.out.println("Car is driving...");
```

```
 }
}
```

In this example, the `Car` class extends the `Vehicle` class, which means it inherits the `start()` method from the `Vehicle` class. Additionally, the `Car` class can have its own unique methods and properties, like the `drive()` method shown above.

Using `implements` and `extends` allows you to create class hierarchies and interfaces that form the basis of object-oriented programming in Java, promoting code reusability and modularity.

These different panes provide various ways to organize and manage the components inside a `JFrame`, allowing you to create complex and visually appealing GUI applications. Depending on your application's requirements, you can utilize these panes to achieve the desired layout and functionality.

————————————————-

Abstract keyword in JAVA

In Java, the `abstract` keyword is used to define an abstract class or an abstract method. An abstract class is a class that cannot be instantiated on its own; it serves as a blueprint for other classes to extend from. An abstract method is a method without implementation, and it must be defined within an abstract class. Abstract classes and methods are marked with the `abstract` keyword.

Here are the key points about the `abstract` keyword:

1. Abstract Class:
 - An abstract class is declared using the `abstract` keyword in its class declaration. - It may contain a mix of abstract methods (without implementation) and concrete methods (with implementation).
 - An abstract class cannot be instantiated directly, meaning you cannot create objects of an abstract class.
 - Subclasses that inherit from an abstract class must provide concrete implementations for all the abstract methods defined in the abstract class.

Example of an abstract class:

```java
abstract class Shape {
 // Abstract method without implementation
 public abstract void draw();

 // Concrete method with implementation
 public void displayArea() {
 System.out.println("Area is calculated based on the shape type.");
 }
}
```

2. Abstract Method:
 - An abstract method is declared using the `abstract` keyword in its method signature. - It does not have a body or implementation in the abstract class where it is declared. - Subclasses that extend the abstract class must provide the implementation for the abstract methods.

Example of an abstract method:

```java
abstract class Animal {
 // Abstract method without implementation
 public abstract void makeSound();
}
```

In this example, `Shape` is an abstract class with an abstract method `draw()`, and `Animal` is an abstract class with an abstract method `makeSound()`. Subclasses extending these abstract classes must provide concrete implementations for the respective abstract methods.

Abstract classes and methods are useful when you want to provide a common base with some shared functionality while leaving specific details to be implemented by subclasses. They play a crucial role in object-oriented design, enabling polymorphism and creating a hierarchy of related classes.

— — — — — — — — — — — — — — -

In Java, you are not required to declare all methods in an interface as abstract explicitly. In fact, all methods declared in an interface are implicitly abstract, meaning they don't have an implementation within the interface itself.

When you define a method in an interface, it's automatically assumed to be an abstract method, regardless of whether you explicitly add the `abstract` keyword or not. This is a fundamental characteristic of interfaces in Java.

Here's an example of an interface with an abstract method:

```java
interface MyInterface {
 void myMethod(); // This method is implicitly abstract
}
```

In this example, `myMethod()` is an abstract method even though the `abstract` keyword is not explicitly used.

Since all methods in an interface are abstract by nature, the interface is essentially a contract that defines a set of behaviors that implementing classes must provide. Any class implementing the interface must override all the abstract methods defined in the interface to fulfill the contract.

In Java 8 and onwards, interfaces can also have default methods and static methods that provide default implementations. Default methods have a concrete implementation within the interface itself, which allows you to add new methods to existing interfaces without breaking backward compatibility with the classes that implement them.

So, to summarize, in Java, you don't need to explicitly declare all methods as abstract in an interface, as they are considered abstract by default. However, you can provide default implementations using default methods if needed.