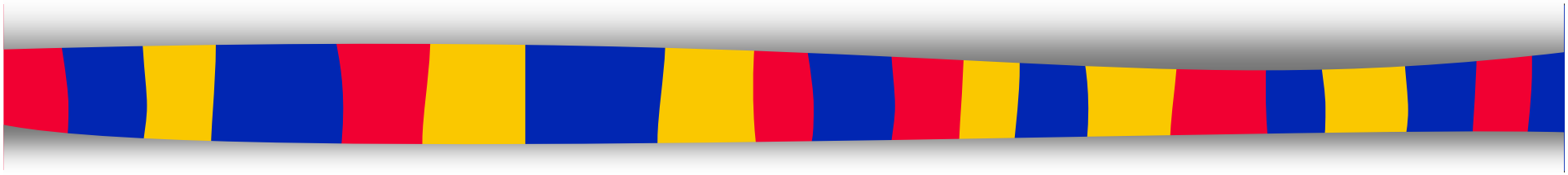
COMP-248
Object Oriented Programming I



Classes and Objects

By Emad Shihab, PhD, Fall 2015,
Parts of the slides are taken from Prof. L. Kosseim
Adapted for Section EE by S. Ghaderpanah, Fall 2015

In this chapter, we will see:

1. Writing our own classes
   - 1.1 Classes and Objects
   - 1.2 Instance Variables
   - 1.3 Methods
2. Some notions of OOP
3. Passing and returning objects
4. Recap

Predefined classes

- Programmers can:
  use pre-defined classes
  develop their own classes

- a *class library* is a collection of pre-defined classes

- the *Java standard class library* (or *Java API* - Application Programming Interface) defines many classes
  ex:
  the **`System`** class
  the **`String`** class

Writing our own classes…

So far, our program had only 1 class with 1 method (**main**)

```
public class MyProgram
{
   public static void main (String[] args)
    { … }
}
```

For large problems…

- the program becomes large and difficult to understand
  e.g., repetition of instructions, variables are dispersed…

Solution:

- Decompose the problem into sub-problems.
- Use methods to implement the sub-problems
- Group related variables and methods into **classes**

1.1 Objects vs classes

Class:

a blueprint/model/pattern to create objects

specified by:

state - descriptive characteristics

behaviors - what it can do (or what can be done to it)

Object:

a specific variable of a class (an instance)

class	**objects**
Date	
Book	
PlayingCard	
Planet	

Examples of classes

Class	State	Behavior
Date		
library book		
bank account		

Classes and Objects

- A class is a type and you can declare variables of a class type.

- A value of a class is called an objects. An object has 2 components:

  - *Data (instance variables)* - descriptive characteristics

  - *Actions (methods)* - what it can do (or what can be done to it)

Objects

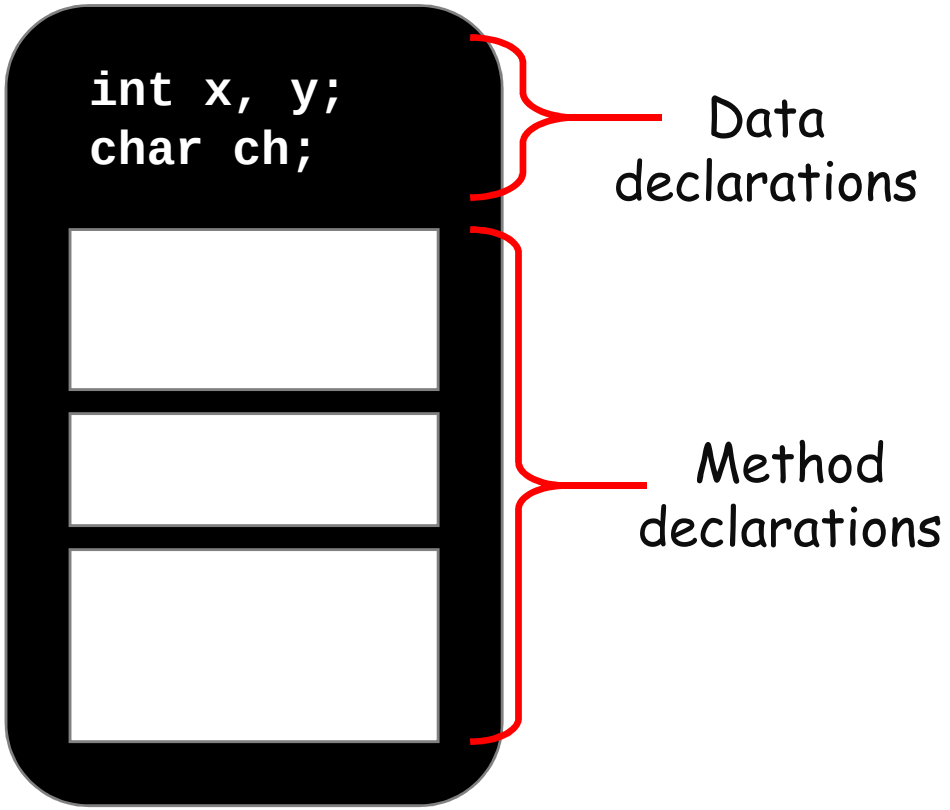
- An object is often referred to as an instance of the class

- Each object can have **different data**, but **all objects have the same actions**.

Declaration of Classes

A class contains:

1. data declarations (instance variables)
2. action declarations (methods)



Example: A class definition

```
public class Date
{
   public String month;
   public int day;
   public int year;

   public boolean isWeekEnd ()
   {
      …
   }

   public void printDate()
   {
      ...
   }
}
```

data declarations (instance variables)

action declarations (methods)

Declaring Objects

- The new operator is used to create an object of a class and associate it with a variable

- Example:
  - `nameOfClass nameOfObject = new nameOfClass();`
  - `nameOfClass nameOfObject ;`
    `nameOfObject = new nameOfClass();`

Example: A driver file

```
public class DateFirstTryDemo
{
   public static void main(String[] args)
   {
      Date date1;
      date1 = new Date();
      Date date2 = new Date();

      date1.month = “December”;
      date1.day = 31;
      date1.year = 2012;
      date1.printDate( );

   }
}
```

declaration of 2 objects

usage of the object

Q: How many instance variable and methods does our object have?

Example: A Date

data (state)

- **day**
- **month**
- **year**

methods (behavior):

- **nextDay**
- **isWeekend**
- **isMyBirthday**
- **printDate**
- …

Classes and files

- In general, we store each class in its own file
- Our example program has 2 files:

1. The **class definition file**:

   defines the data and methods of a class

2. The **driver file**:

   contains the **`main`** method

   declares and uses objects of the class
   controls the use of other parts of a program
   often used to test other parts of the program

Just checking …

The new operator:

A. allocates memory
B. is used to create an object of a class
C. associates an object with a variable that names it.
D. all of the above.

In this chapter, we will see:

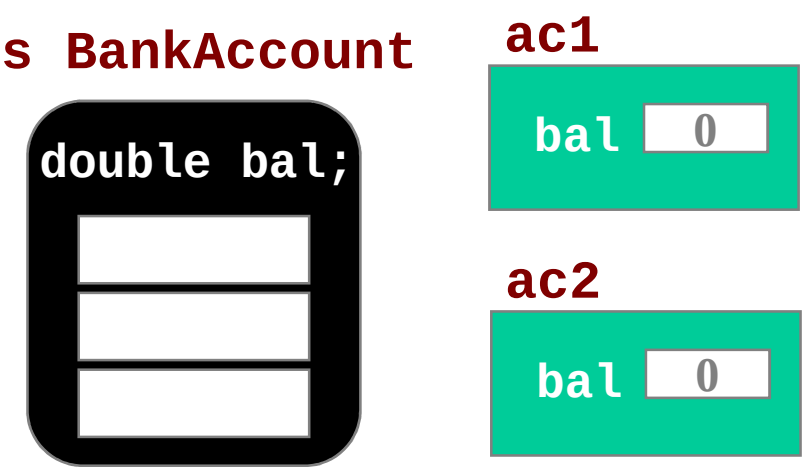
1. Writing our own classes
   - 1.1 Classes and Objects
   - 1.2 **Instance Variables**
   - 1.3 Methods
2. Some notions of OOP
3. Passing and returning objects
4. Recap

Instance Variables

- variables/constants declared inside the class **but not inside a specific method**
- also called attributes
- can be used by any method of the class
- initialized to 0 (false for booleans)
- each object (instance) of the class has its own instance data

Example:

```
BankAccount ac1=new BankAccount();
BankAccount ac2=new BankAccount();
```



In this chapter, we will see:

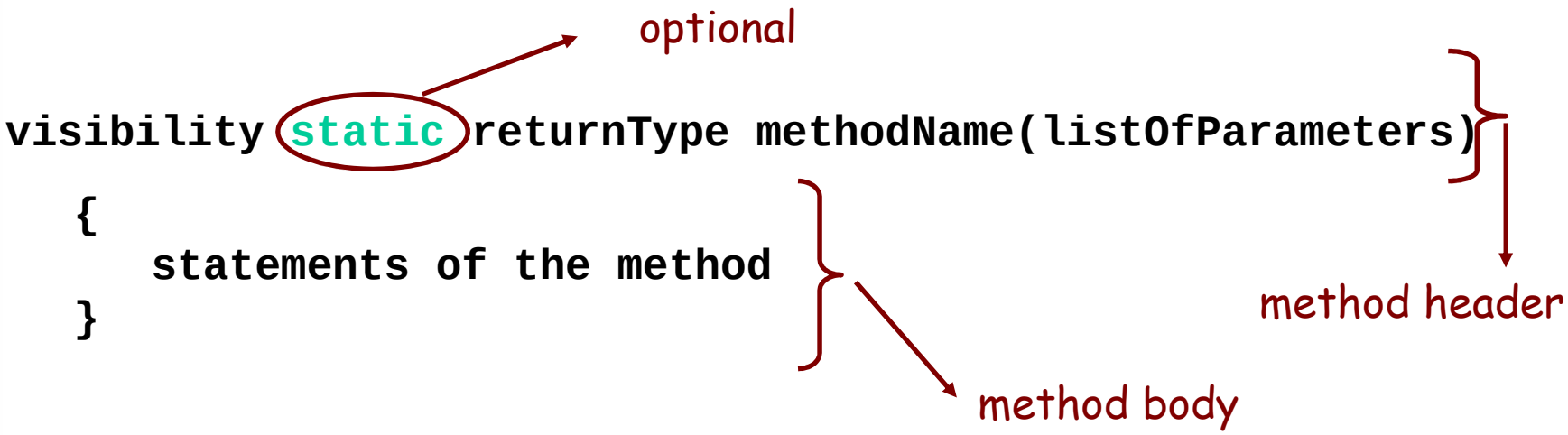
1. Writing our own classes
   - 1.1 Classes and Objects
   - 1.2 Instance Variables
   - 1.3 **Methods**
2. Some notions of OOP
3. Passing and returning objects
4. Recap

Methods

- Implement the behavior of all objects of a particular class
- All objects of a class share the method definitions
- Some methods are a bit special…
  (ex: constructor)
- Group of statements that are given a name
- A method is defined once, but can be used (called/invoked) several times

Definition of a method

- Method header and method body




```
public void sayHi()
{
  System.out.print("Hi");
}
```

```
public static void main(String[] args)
{
   …
}
```

Methods cont'd…

There are two kinds of methods:

- Methods that compute and return a value
- Methods that perform an action
  does not return a value
  is called a **`void`** method

Methods that return a result

- Must specify the type of that value in its heading:
  `public typeReturned methodName(paramList)`

- Examples:

  description: determines if the coin is a tail (1==tail, 0==heads)
  name: **isTail**
  result: **boolean**

```
public boolean isTail()
{
  if (face == 1)
     return true;
  else
     return false;
}
```

  description: returns the value of the face
  name: **getFace**
  result: **int**

```
public int getFace()
{
   return (face);
}
```

The return statement

Allows the method to "return" a result

syntax: **`return expression;`**

1. the expression is evaluated
2. the value of the expression is returned as the result of the method
3. the method is exited

```
public boolean isTail()
{
  if (face == 1)
     return true;
  else
     return false;
}
```

```
public int getFace()
{
   return (face);
}
```

```
public boolean isTail()
{
  return (_____________________________);
}
```

Another way of writing the method **`isTail()`** is …. ?

Methods that return no result

they perform an action performed, but do not "evaluate" to a value

ex: they display something, change the value of an attribute, …

They officially return **void**

They use no **return expression;**

or: **return;**

```
public void flip()
{
   face = (int)(Math.random()*2);
}
```

```
public _____ printFace()
{
  System.out.print(face);
}
```

```
public void flip()
{
    face = (int)(Math.random()*2);
    return;
}
```

Next class, we will see:

1. Writing our own classes
   - 1.1 Classes and Objects
   - 1.2 Instance Variables
   - 1.3 **Methods (More)**
2. Some notions of OOP
3. Passing and returning objects
4. Recap

Accessing members

to access any members (data or method)

within the class

```
nameOfData
nameOfMethod(actualParameters)
```

from outside the class

non-static:

```
nameOfObject.nameOfData
nameOfObject.nameOfMethod(actualParameters)
```

static:

```
nameOfClass.nameOfData
nameOfClass.nameOfMethod(actualParameters)
```

Example

```
public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;
   private int face;

   public void flip()  {
      _______= (int)(Math.random()*2);
   }

   public boolean isHeads() {
      return (_______ == HEADS);
   }

  // flips the coin 5 t
   public void flip5() {
      for (int i=1; i<=5; i++)
         ___________________
   }
…

}
```

class def.

Example

```
public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;
   private int face;


   public void flip()  {

      _______= (int)(Math.random()*2);

   }


   public boolean isHeads() {

      return (_____________ == HEADS);

   }


  // flips the coin 5 times
   public void flip5() {

      for (int i=1; i<=5
          _____________

   }
```

class def.

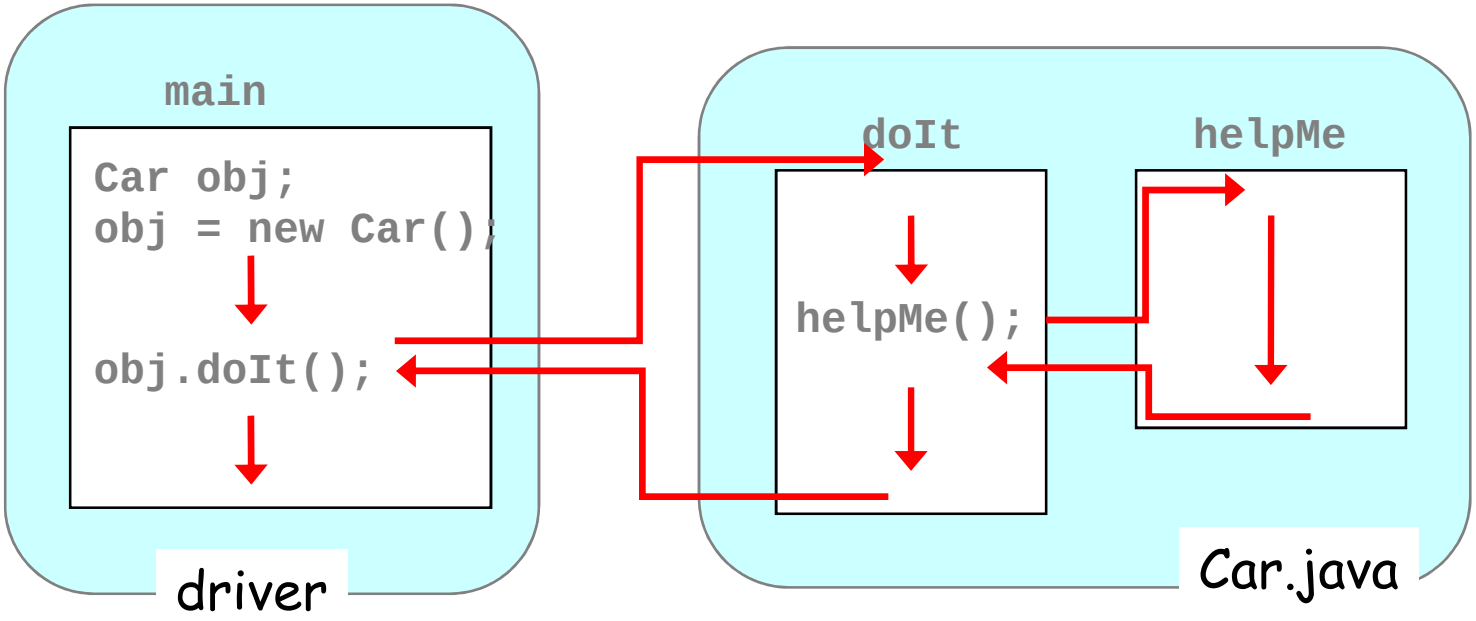
```
_____________________________________;
_____________________________________;
    coin1.isHeads()
    coin2.isHeads()

flip();
coin1.flip;
```

driver

Method control flow

When a method is invoked:

1. the current method is suspended
2. the called method is executed
3. when completed, the flow returns to the place where the method was called and resumes its execution



Calling methods

methods that return a result are expressions that have:

a type and a value

methods that do not return a result are **not** expressions

you call them with the statement: `objectName.methodName();`

```
coin1.flip();
        coin1.flip();
    coin1.flip()

                coin1.flip()
```

driver

Just checking ….

Which one of these statements is syntactically correct?

```
        Coin coin1 = new Coin();
        boolean result;
/* 1 */ coin1.isTail();
/* 2 */ result = coin1.isTail();
/* 3 */ if (coin1.isTail())
          System.out.print("whatever");
/* 4 */ System.out.print(coin1.isTail());
```

A. All are syntactically correct
B. 2 and 3 only are syntactically correct
C. 2, 3 and 4 only are syntactically correct
D. 1 is the only one which is syntactically correct
E. 3 is the only one which is syntactically correct

DateSecondTry.java

DemoOfDateSecondTry.java

Just checking …

The body of a method that returns a value must contain at least one _____ statement.

A. void
B. invocation statement
C. declaration
D. return

Local / Global Variables

Local variables:

- declared inside a method
  - variable that is necessary for that method only
- method parameters are considered local variables

- If 2 methods have a local variable of the same name, they are 2 entirely different variables

Global variables:

- Java does **not** have global variables

Parameters

Some methods need to receive additional data in order to perform their work

allows the function to be more generic

ex. **`sqrt`** method works for any double

```
Math.sqrt(9), Math.sqrt(15.5),
Math.sqrt(75),...
```

definitions:

formal parameter:

parameter specified in the method header

argument or actual/run-time parameter:

parameter specified in the method call / invocation

Example 1

```
public class Account
{
   private double balance;

   public void deposit (double theAmount)
   {
      if (theAmount < 0)  // deposit value is negative
         System.out.println ("Error: Deposit amount is invalid.");
      else
         balance = balance + theAmount;
}
```

class def.

```
25.85
10
```

driver

DateThirdTry.java

DateThirdTryDemo.java

Example 3

```
public class Store
{
   public void printPrice(_____________________ , _____________________ )
   {
     System.out.println ("the " + _________________ + " costs " + _____________);
   }
 …
```

class def.

```
"desk", 300.99

"chair", special


  s, Math.min(100, p)
```

driver

Exercise

in the class **Account**, write the header of the methods:

```
    // withdraw
     _____________________________________________________________
    // getInterestRate
     _____________________________________________________________
    // changeName
     _____________________________________________________________
    }
}
```

Exercise cont'd

Just checking …

A variable whose meaning is confined to a method definition is called an/a

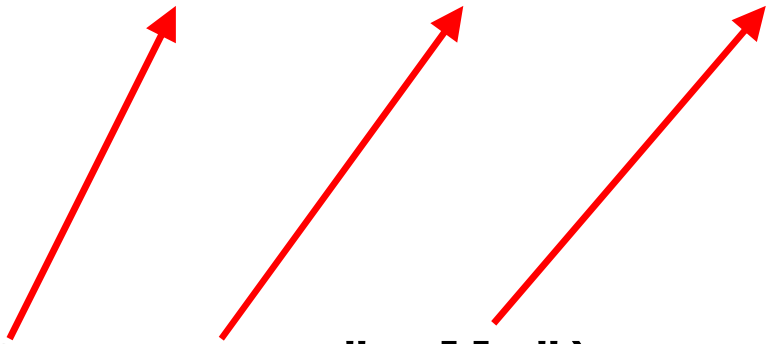
A. instance variable
B. local variable
C. global variable
D. none of the above

Call by value

when a method is called:

- the actual parameters are **copied** into the formal parameters
- the method works with a **copy** of the actual parameters
- … when we return from the function, the actual parameters are **unchanged**

```
public void someMethod(int num1, int num2, String message)
{
   …
}
```



```
someMethod(25, count, "Hello");
```

Example

```
public class AClass
{
 private double anAttribute;

 public void aMethod(int aParam)
 {
   System.out.println(aParam);
   if (aParam < 0)
      aParam = 0;
   anAttribute = aParam;

   System.out.println(aPar
 }
```

class def.

driver

output

Formal and Actual parameters

when a method is called:

the order of the actual param. must be == to the order of the formal param.

the nb of actual param. must be == to the nb of formal param.

the types of the actual param. must be compatible with the types of the formal param.

```
aMethod
```

class def.

```
aMethod
aMethod
aMethod
aMethod
aMethod
```

driver

Type Conversion of Parameters

the types of the actual param. must be compatible with the types of the corresponding formal param.

```
public double myMethod(int p1, int p2, double p3) {…}
…
int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```

If no exact type match --> automatic type conversion

c is casted to a **double**

remember:

byte→short→int→long→float→double

char

The `this` Reference

All instance variables are understood to have `<the calling object>.` in front of them

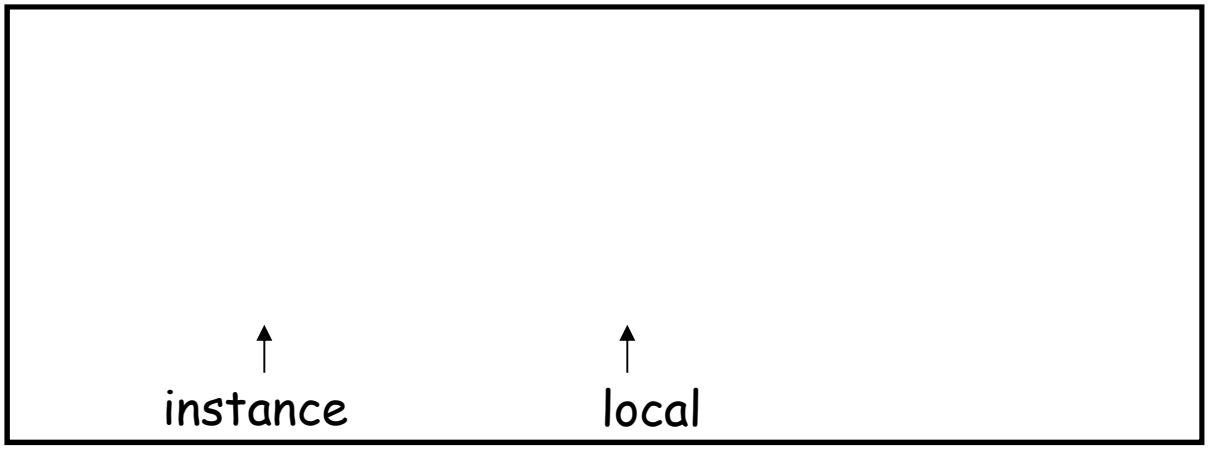
inside a method:

`myInstanceVariable` is identical to `this.myInstanceVariable`

The `this` Reference

`this` must be used:

- if a parameter or other local variable has the same name as an instance variable
- otherwise, the local variable will be used



A special method: `toString()`

when an entire object is printed…

```
System.out.println(acct1);
```

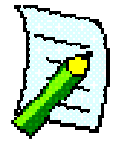
driver

the **`toString`** method is automatically called

if you have not defined a **`toString`** method, then

otherwise, define your own version of **`toString`** for your object

Defining your own **`toString()`**



-
-

```
public  _____________________toString(             ) {
   return (acctNumber + "\t" + name + "\t" + balance);
}
```

class def.

Another special method: `equals()`

to compare two objects, you can use:

- `==` operator
  - compares equality of references
  - returns **`true`**:
    - if the references point to the same object
    - NOT if the objects pointed have the same content

- equals method
  - compares equality of the content of the objects
  - returns **`true`**:
    - if the references point objects that have the same content

Equality of objects

The method **`equals`** :

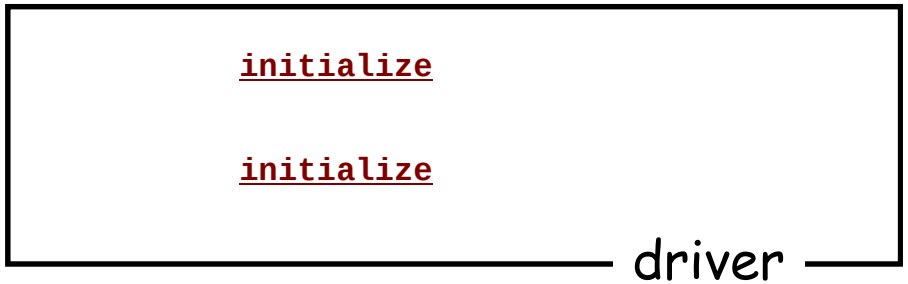
- is defined for all objects
- unless we redefine it in our class, it has the same semantics as ==
- we can redefine it to return **`true`** under whatever conditions we think are appropriate (ex. equality of content, not address)

Example

EqualsAndToStringDemo.java

DateFourthTry.java

Constructors



Constructors

-
-
- same name
- no
- **public**
-

Example 1

```
public class BankAccount
{
   private long acctNumber;
   private double balance;
   private String name;

   // a constructor:
   public _____( String theOwner,
                 long theAccount,
                 double theInitial)

   {
      name = theOwner;
      acctNumber = theAccount;
      balance = theInitial;
   }

   public void deposit(double amount)
   { … }
   public void withdraw(double amount)
   { … }
   …
```

BankAccount.java

Example 2

```
public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;
   private int face;

   public Coin()
   {
      flip();
   }

   public void flip()
   {
      face = (int)(Math.random()*2);
   }

   public boolean isHe
   {
      return (face == HEADS);
   }
```

Coin.java

```
Coin myCoin = new Coin();
Coin anotherCoin = new Coin();
Coin athirdCoin = new Coin();
```

driver

Example 3

Date.java

ConstructorsDemo.java

Default Constructor

If you do not include any constructors in your class,

- Java will automatically create a *default* or *no-argument* constructor

The default constructor:

- takes no arguments
- initializes all instance variables to zero (null or false)
- but allows the object to be created

If you include even one constructor in your class

- Java will not provide this default constructor

Overloading methods

overloading = same name is used to refer to different things

- "chair" (person or furniture)
- / (integer or real division)

overloaded methods:

- several methods have the same name
- but different definitions

the ***signature*** of each overloaded method must be unique

- signature = name of method + parameter list
- the compiler determines which version of the method is being invoked by analyzing the signature
- the return type of the method is <u>not</u> part of the signature

Example

Version 1

```
public int increment(int x)
{
   return x+1;
}
```

Version 2

```
public int increment(int x, int value)
{
   return x+value;
}
```

Invocations

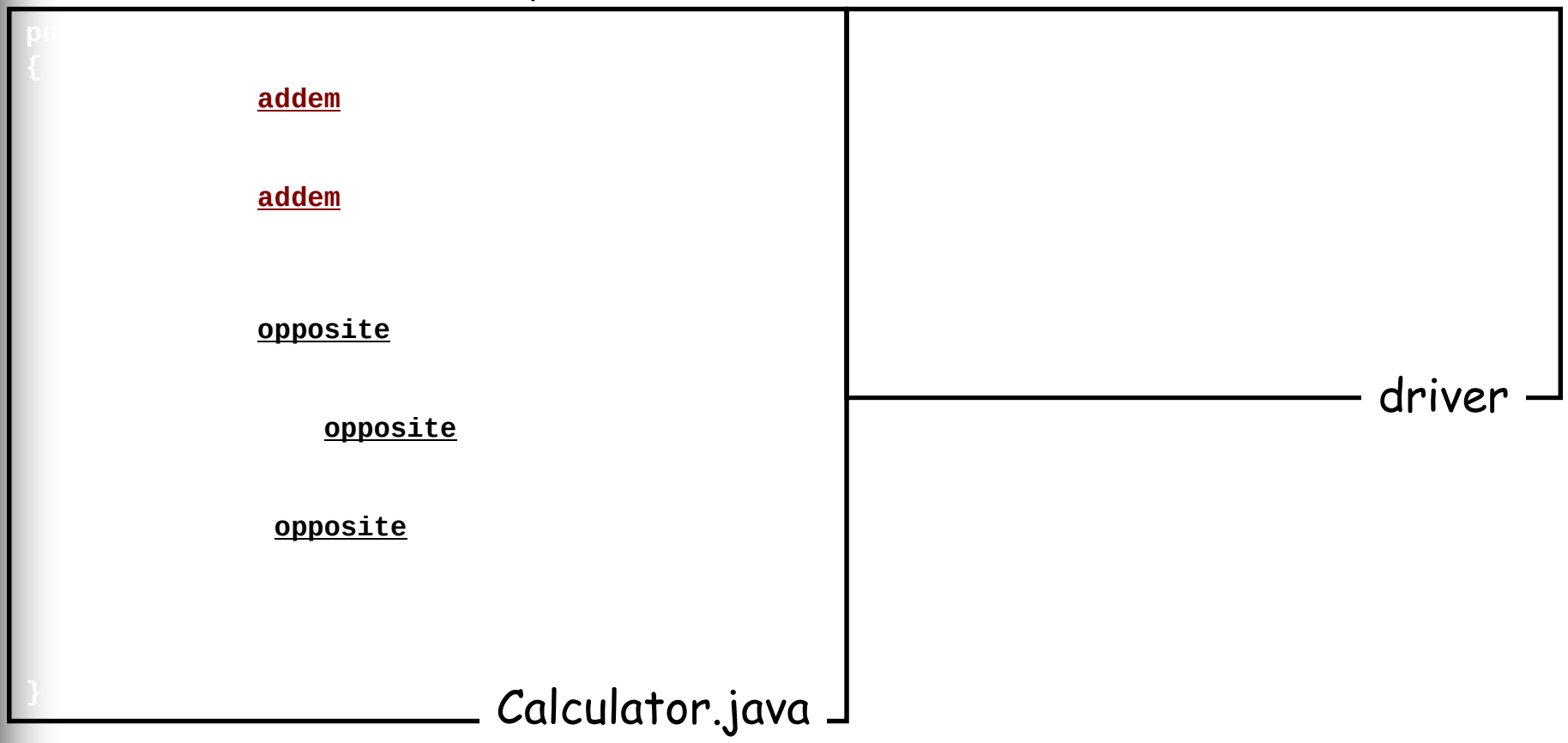
```
int result = 1;

result = increment(20, 4);     // 1. OK?
result = increment(10);        // 2. OK?
result = increment(result);    // 3. OK?
result = increment(result, result);  // 4. OK?
result = increment(result, result, result); // 5. OK?
result = increment(result, increment(result, result));// 6. OK?
result = increment();          // 7. OK?
```

Overloaded methods

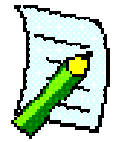
guess what… the **`println`** method is overloaded!

Overloaded methods

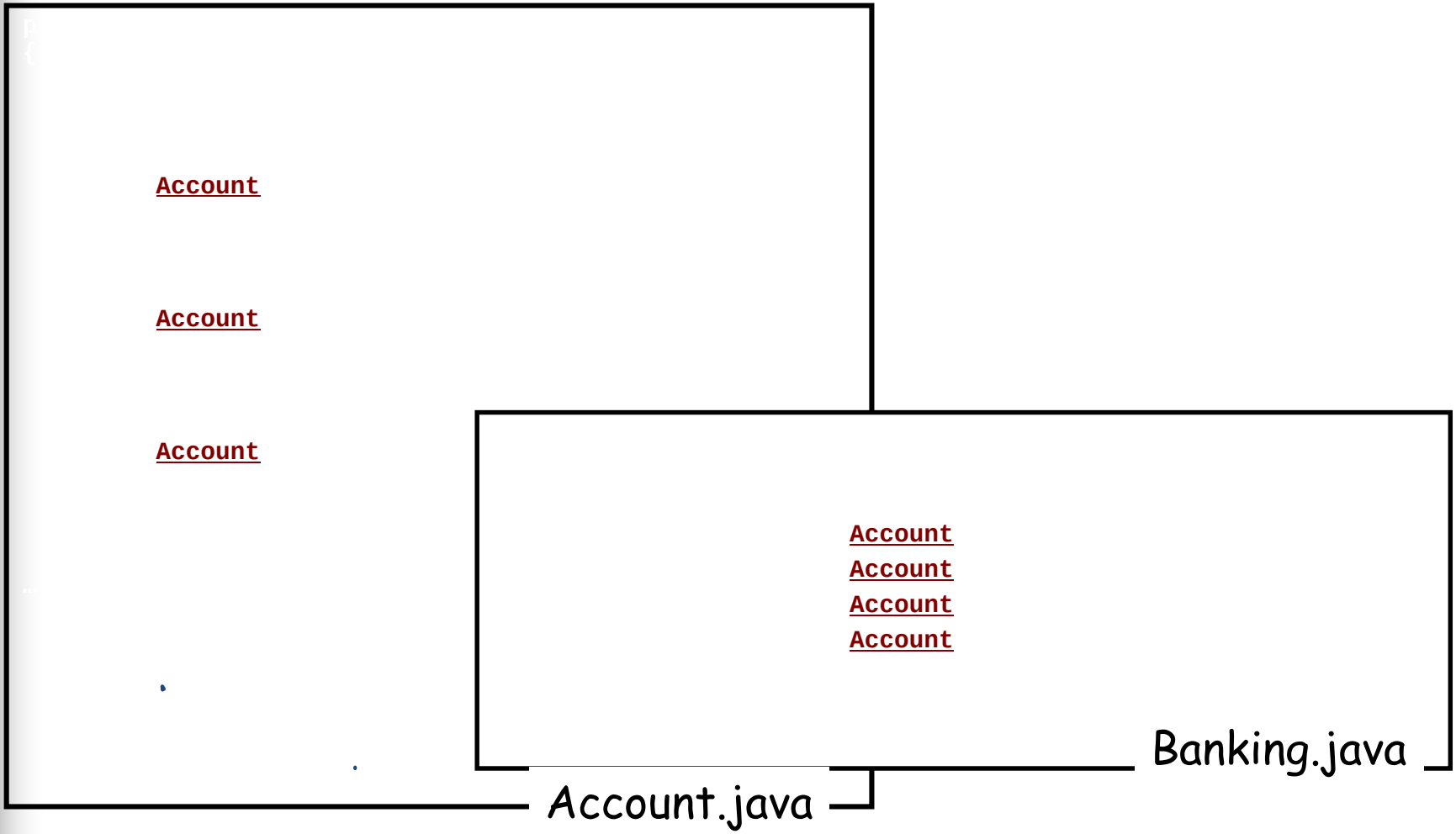
practical when the same operation must be done on different types or different numbers of parameters



Overloading constructors



Constructors are methods… so they can be overloaded



In this chapter, we will see:

1. Writing our own classes
   - 1.1 Objects vs classes
   - 1.2 Instance Variables
   - 1.3 Methods
2. Some notions of OOP
3. Passing and returning objects
4. Recap

2- Some notions of OOP

Information hiding

separating how to use a class from how it is implemented

Abstraction is another term used

Encapsulation

the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details

Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface

In Java, hiding details is done by marking them **`private`**

Interface

an object interacts with the rest of the program via an interface

interface = set of methods that allow access to the object

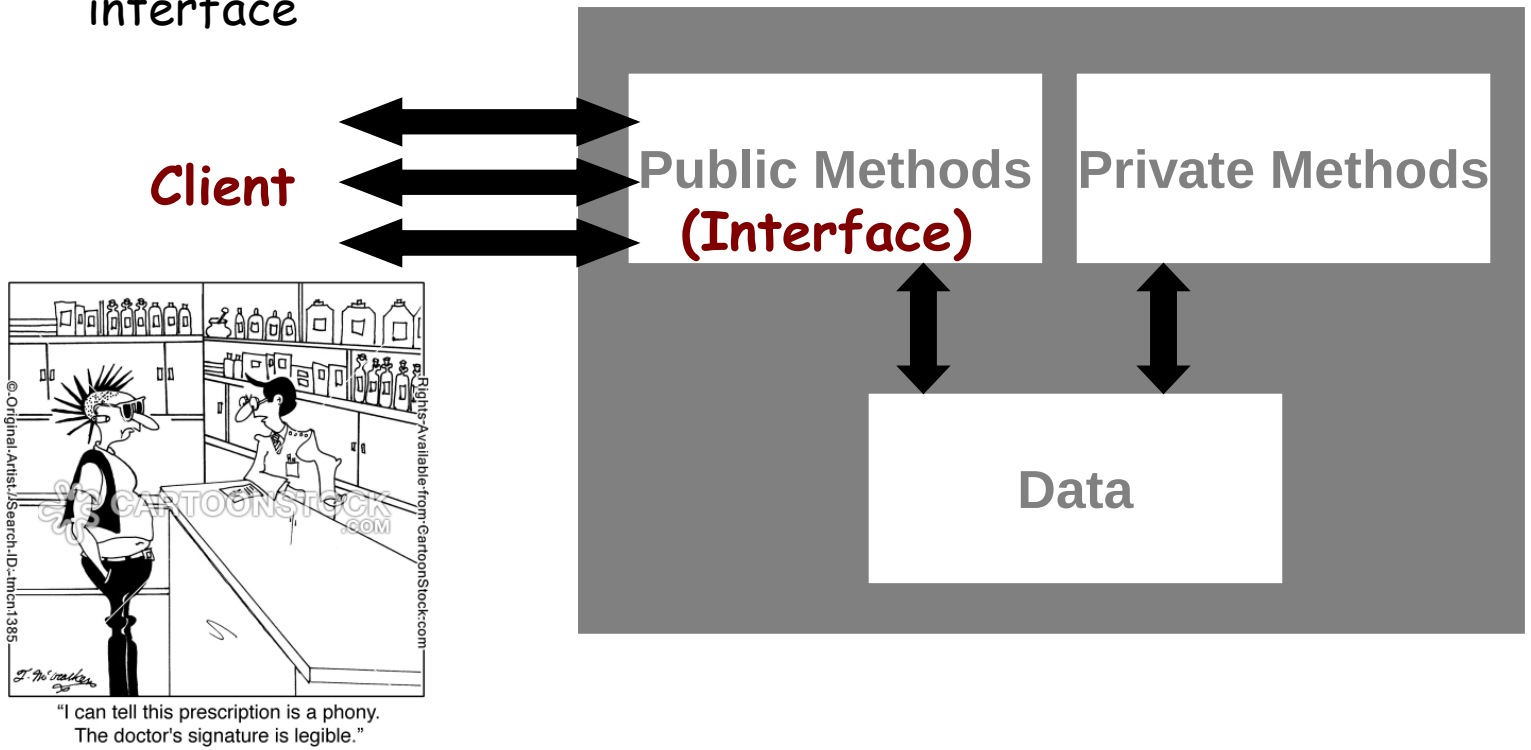
the interface hides how a method is implemented

Encapsulation

An encapsulated object can be thought of as a *black box*

its inner workings are hidden to the client

the client can access the object only by invoking the methods of the interface



"I can tell this prescription is a phony.
The doctor's signature is legible."

Visibility modifiers

Java has 4 visibility modifiers:

- **`public`**
- **`private`**
- **`protected`** (involves inheritance, see COMP 249)
- **`default or package`** (COMP 249)

can be applied to all members (data and methods)

Visibility modifiers

public members
- can be directly accessed from anywhere (inside & outside the object)
- violate encapsulation

private members
- can only be accessed from inside the class definition

by default…
- members can be accessed by any class in the same package
- i.e. access is more open than private, but more strict than public

Visibility modifiers

Data:

- should be **private**
- public data violate encapsulation
- constants are OK to be public because they cannot be modified anyways

Methods:

- should be **public**
  - if the method provides the object's services
  - so that it can be invoked by clients
  - also called ***service method***
- should be **private**
  - if the method is created simply to assist a service method
  - also called a ***support method***

Visibility modifiers

	`public`	`private`
Variables	NO! Violates encapsulation	YES! Enforces encapsulation
Methods	Yes, if method provides services to clients	Yes, if method supports other methods in the class

Example

```
public class Account {
  private double rate;
  private long acctNumber;
  private double balance;
  private String name;

  public Account(String owner, long nb,
    double init) {
     name = owner;
     acctNumber = nb;     //  1. OK?
     balance = init;
     rate = 0.02;
  }
  public double deposit(double amount)   {
     if (amount < 0)  // deposit is negative
       System.out.println("Error");
     else
        balance += amount; // 2. OK?
     return balance;
  }
  private void changeRate(double newRate) {
     rate = newRate;
```

Account.java

```
acct1.deposit
acct1.balance
        acct1.deposit
    changeRate
```

driver

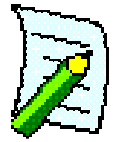
Example

```
public class Printer
{
  private final int DEFAULT_NB = 10;

  public void printMany(int nbTimes, String theMessage)
  {
    if (nbTimes < 0 || nbTimes > 100)
        internalUseOnly(DEFAULT_NB, theMessage);
    else
       internalUseOnly(nbTimes, theMessage);
  }

  private void internalUseOnly(in
  {
    for (int i                    imes;
       System.out.println(AMess);
  }
}
```

class def.

```
driver
```

Accessor and Mutator methods



data members are usually private

so to access them, we usually have *set* and *get* methods

mutator (setX): sets the data X and makes sure it stays in a coherent state

accessor (getX): returns the value of data X

```
setHour

setMinute

setSecond
```

```
getHour

getMinute

getSecond
```

Mutators that return a boolean

if a mutator is given a value that would make the object in an invalid state, it can:

1. display an error message, and quit the program, OR
2. return a boolean (false), and let the calling method decide what to do

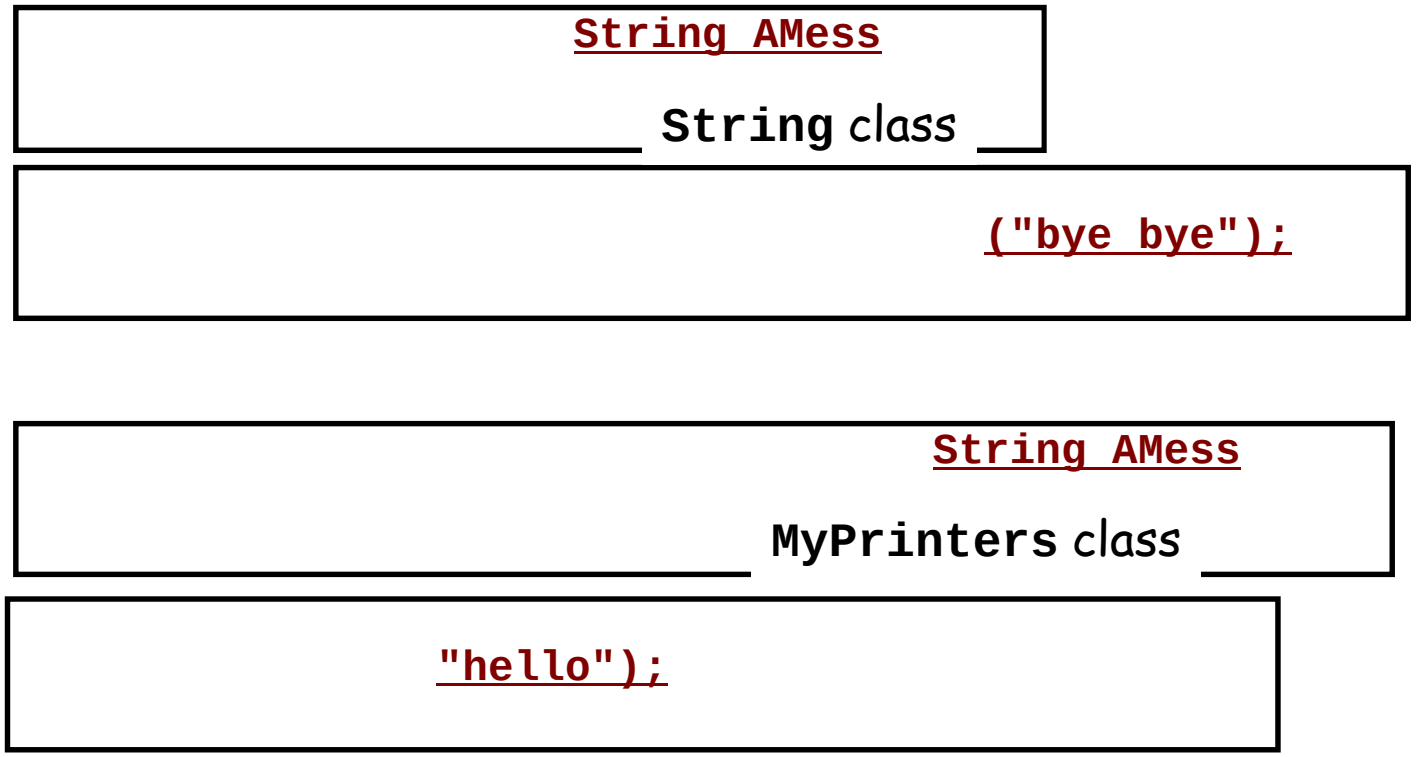
In this chapter, we will see:

1. Writing our own classes
   - 1.1 Objects vs classes
   - 1.2 Instance Variables
   - 1.3 Methods
2. Some notions of OOP
3. Passing and returning objects
4. Recap

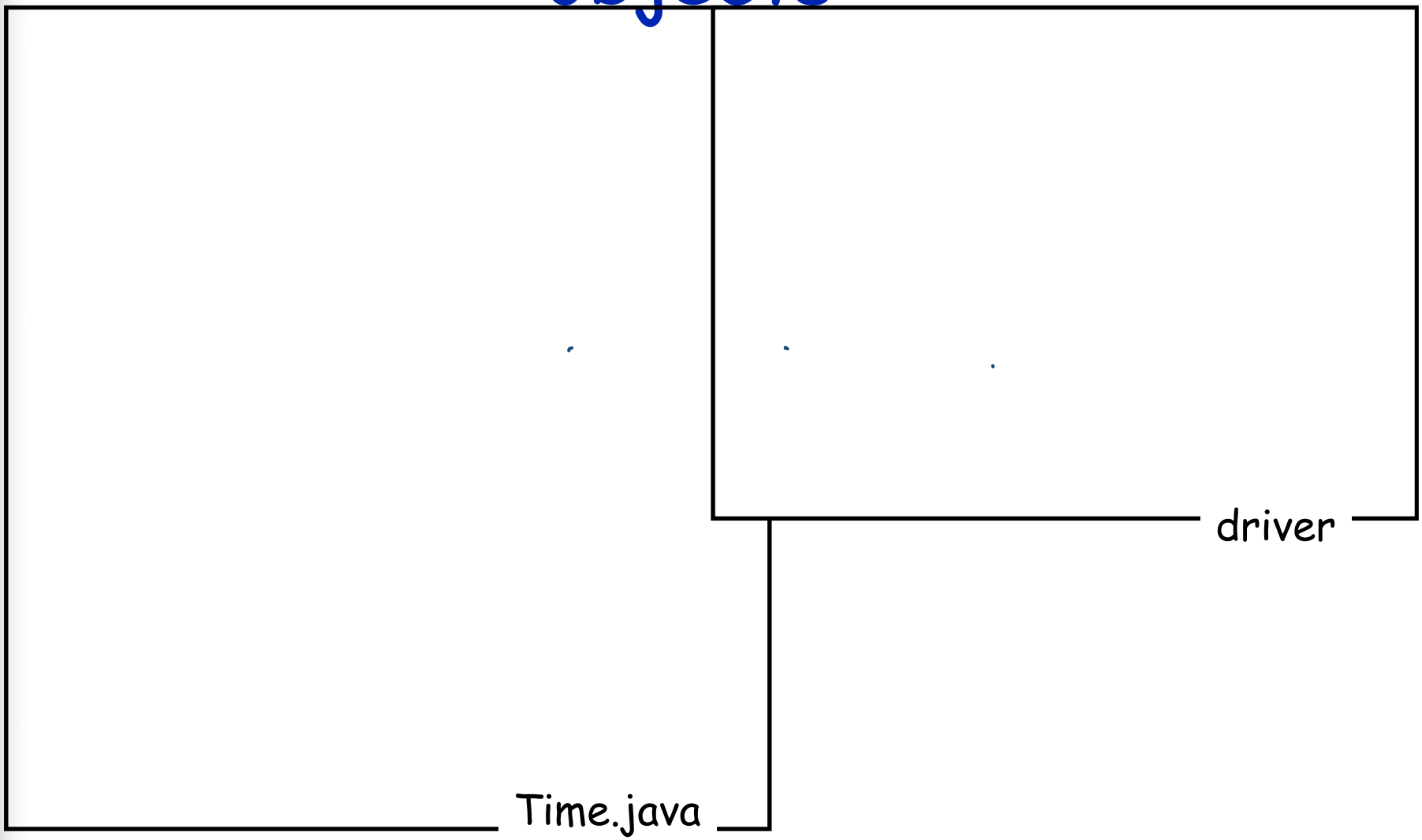
3- Passing and returning objects

an object can be a parameter to a method

ex:



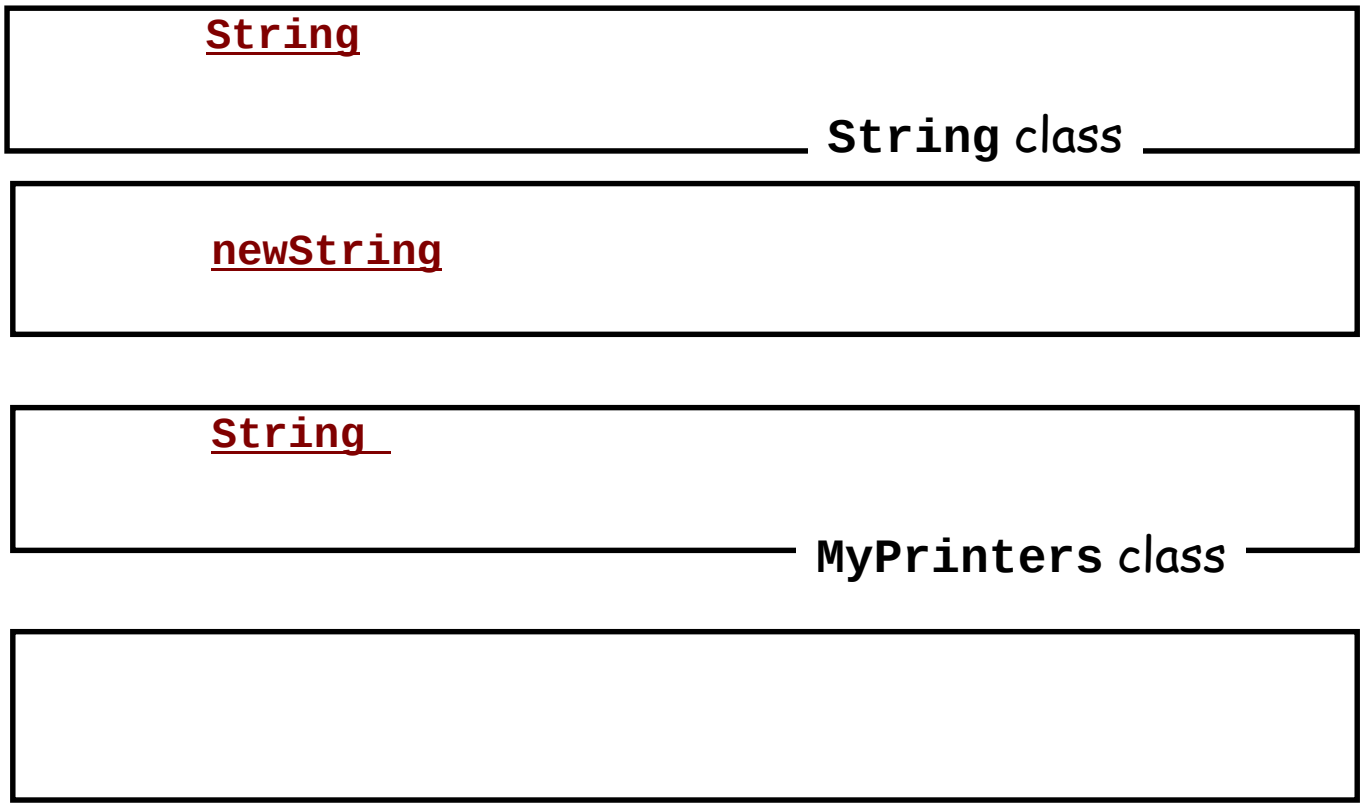
Example: compare 2 `Time` objects



Passing and returning objects

an object can be returned by a method

ex:



Example: add 2 `Time` objects

```
result.hour = hour + t2.getHour();
result.minute = minutes + t2.getMinute();
```

driver

Time.java

In this chapter, we will see:

1. Writing our own classes
   - 1.1 Objects vs classes
   - 1.2 Instance Variables
   - 1.3 Methods
2. Some notions of OOP
3. Passing and returning objects
4. Recap

4- Recap

- to define a class:
  - data members (attributes)
    - declared outside any method
    - must decide:
      - visibility (private? public?)
  - methods
    - must decide:
      - visibility (private? public?)
      - type of result (void?, int?, boolean?…)
      - number and types of parameters
      - actual code of the method
      - static or not? (so far… never static)
- 2 files:
  - the class definition (ex. Coin.java, BankAccount.java)
  - the driver program (ex. CoinFlip.java, Banking.java)

Just checking …

A set method is:

A. an accessor method
B. a mutator method
C. a recursive method
D. none of the above

Just checking …

Accessor methods:

A. return the value of an instance variable
B. promotes abstraction
C. both A and B
D. none of the above

Just checking …

`this` refers to:

A. instance variables
B. local variables
C. global variables
D. the calling object

Just checking …

The name of a method and the list of _____ types in the heading of the method definition is called the method signature.

A. parameter
B. local variable
C. return
D. primitive

Let's put it all together: rational numbers ($\frac{7}{8}$)

name	type	visibility
numerator		
denominator		

name	return type	parameters	visibility
Rational			
getNumer			
setNumer			
reciprocal			
add			
toString			
reduce			

```
public class Rational {
  // data members:

 // methods
                                        {…}
                                        {…}
                                        {…}
                                        {…}
                                        {…}
                                        {…}
                                        {…}
}
```

In this chapter, we have seen:

1. Writing our own classes
   - 1.1 Objects vs classes
   - 1.2 Instance Variables
   - 1.3 Methods
2. Some notions of OOP
3. Passing and returning objects
4. Recap