COMP9103: Software Development in Java

W4: Object-Oriented Programming

Presented by
Dr Ali Anaissi
School of Computer Science

THE UNIVERSITY OF SYDNEY

Object-Oriented Paradigm

Object-Oriented Paradigm

- **The real world** consists of different classes of objects (humans, animals, cars, …)
- Each **class** of objects has **common characteristics** and can perform some **common actions/behaviors**:
  - For instance, cars have **characteristics** such as:
    - Hand wheel,
    - doors,
    - brake,
    - 4 wheels ,
    - …
  - **Common actions** for cars, for example,
    - Speed up,
    - Slow down,
    - Open the door,
    - …

Object-Oriented Paradigm

- The real world can be modeled by **Object-Oriented Programming**
- In object-oriented programming, we write Java code to create new data types with:
  - More complex data structures
  - Different methods or functions for manipulating and processing the data
- OOP enables us to
  - create our own data types (e.g., `Car`)
  - define operations in the data types (e.g., `accelerate`, `turn`, ...)
  - and integrate them into our programs (e.g., `a racing game`).
- OOP:
  - Breaks a complex problem into small manageable components called ***classes***
  - enhances the reuse of software components without re-programming
  - Reduces the costs, time, and workload of software development, testing and maintenance

Classes

- **Class:** provides a **blueprint** for defining and creating objects
- A class definition includes:
  - **states** (as **fields/variables**)
  - **behaviors** (as **methods**) of all the objects of that class
  - **constructors** to specify what needs to be done (e.g., initialize some fields / variables) when a new object is created

Class: Car

Attributes/Data/Variables/Fields	Behaviors/Methods
Color	Accelerate: press gas pedal
Brand	Decelerate: press brake pedal
Owner	…
…	

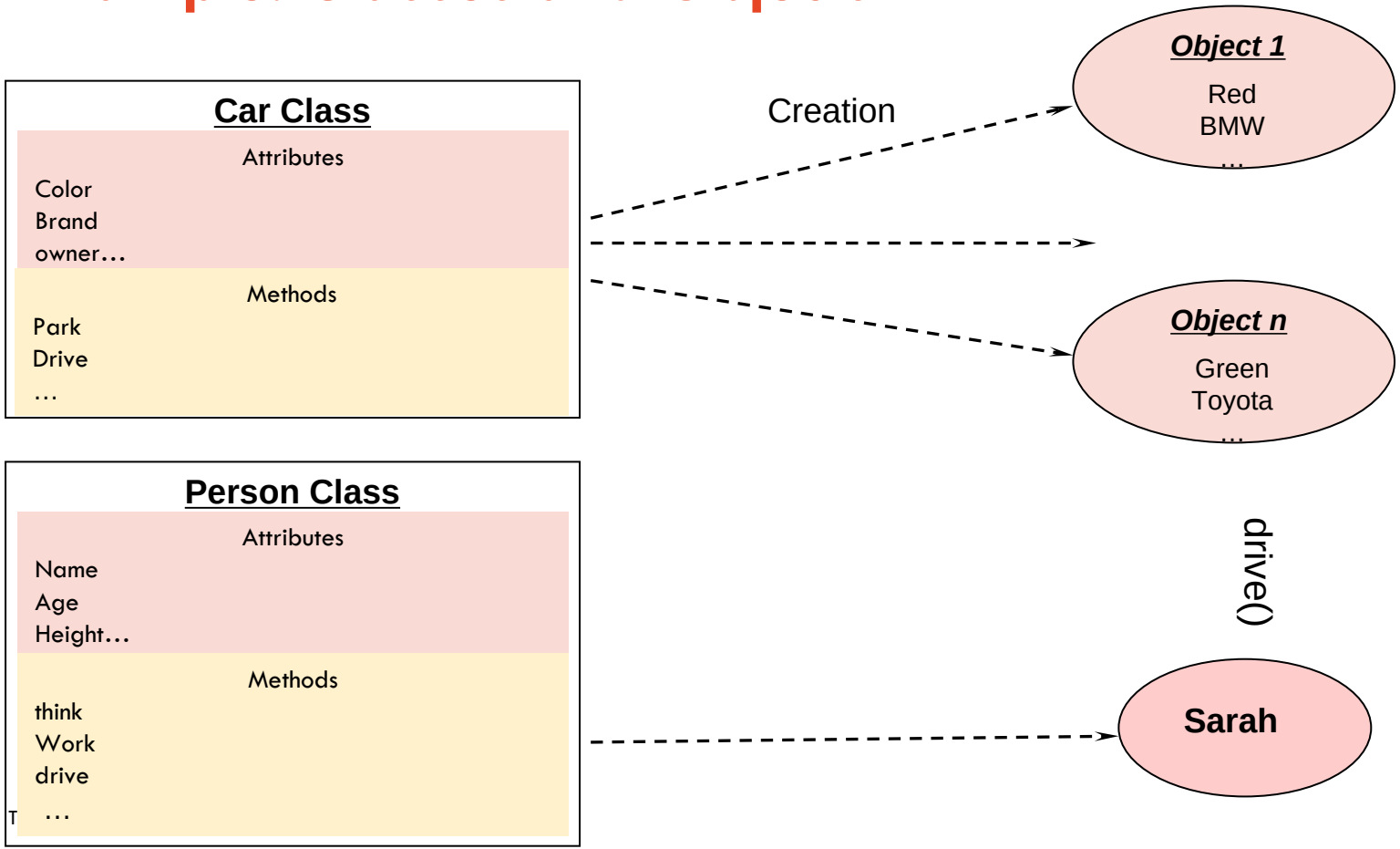
Class definition does **NOT reserve any memory** for the instance variables!

Objects

- **An object is a specific instance of a class**
  - e.g. Tom's car

- **An object has data and methods to represent its states and behaviors**
  - **Instance fields/variables:**
    - Describe the states/attributes of a specific object
  - **Methods:**
    - specify what an object can do
    - usually operate on the fields of an object
    - may change an object's attributes/states
    - enable an object to react to its environment
- **Objects are entities in your program, which are manipulated by calling methods.**

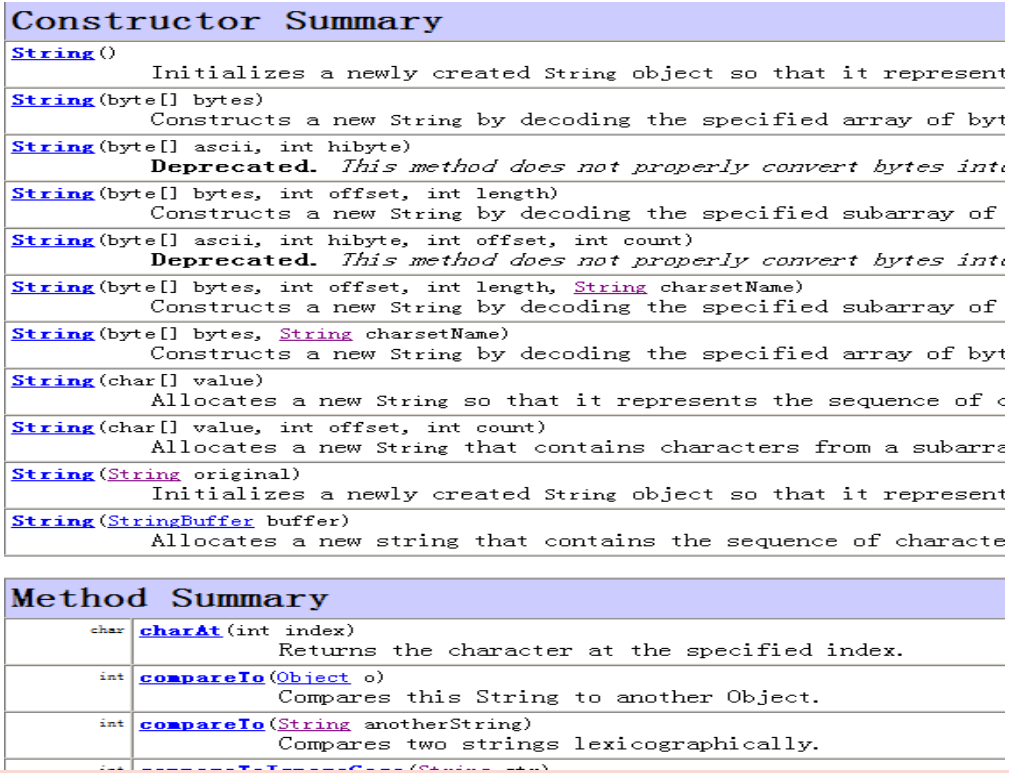
Example: Classes and Objects



String Class

THE UNIVERSITY OF SYDNEY

String Class



Constructor Summary

String()	Initializes a newly created String object so that it represent
String(byte[] bytes)	Constructs a new String by decoding the specified array of byt
String(byte[] ascii, int hibyte)	**Deprecated.** *This method does not properly convert bytes int*
String(byte[] bytes, int offset, int length)	Constructs a new String by decoding the specified subarray of
String(byte[] ascii, int hibyte, int offset, int count)	**Deprecated.** *This method does not properly convert bytes int*
String(byte[] bytes, int offset, int length, String charsetName)	Constructs a new String by decoding the specified subarray of
String(byte[] bytes, String charsetName)	Constructs a new String by decoding the specified array of byt
String(char[] value)	Allocates a new String so that it represents the sequence of c
String(char[] value, int offset, int count)	Allocates a new String that contains characters from a subarra
String(String original)	Initializes a newly created String object so that it represent
String(StringBuffer buffer)	Allocates a new string that contains the sequence of characte

Method Summary

char	**charAt**(int index) Returns the character at the specified index.
int	**compareTo**(Object o) Compares this String to another Object.
int	**compareTo**(String anotherString) Compares two strings lexicographically.

http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

The `String` Class and its Objects

`String` class – a blueprint

- Data: a string of characters
- Methods:
  - **length**(): Returns the length of this string.
  - **charAt**(int index): Returns the char value at the specified index.
  - **equals**(Object anObject): Compare this string to the specified object.
  - … …

an instance/object of String referred by ***str:*** a concrete object

Constructed with

String str = new String("Hello, World!")

Data: Hello, World!

Methods:

- `str.length()`: Returns 13.
- `str.charAt(4)`: Returns 'o'.
- `str.equals("hi")`: false.

Primitive Types and Reference Types

- In Java, no variable can ever hold an object!
- A reference variable can only hold a reference (memory location / address) of an object
- Every variable represents a memory location that holds a value.
  - For a variable of primitive type, the value is of the primitive type
  - For a variable of reference type, the value is a reference to where an object is located.

Primitive type

```
int i=1;
```

Reference type

```
String s1
```

```
new String("Hello");
```

Scanner Class

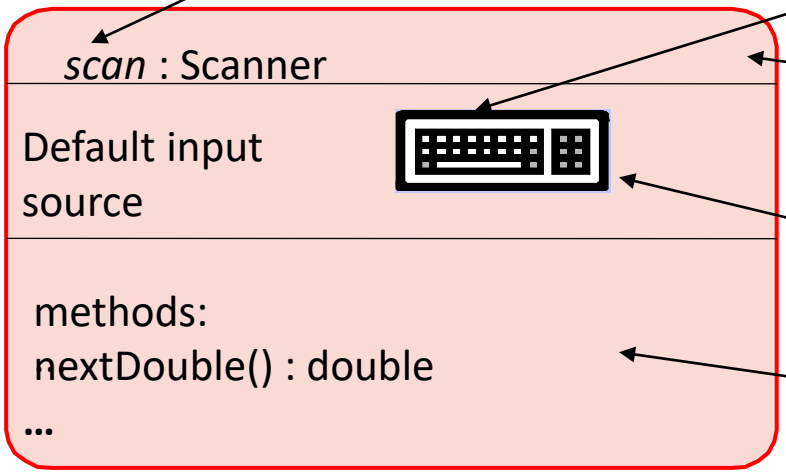
Scanner Class

- It is a part of the standard Java class library, *provides convenient methods for reading input values of various types*

- The input could come from different sources, including:
  - data input interactively by the user
  - or data in a file

Interactive input via Scanner class

- **Scanner** class
  - Makes input from the keyboard easy
- `Scanner scan = new Scanner(System.in);`



Reference variable *scan* accesses to an input source

input source for this Scanner object is the keyboard

Scanner supports a large number of methods for inputs

Interactive Input via Scanner

```
import java.util.Scanner;
public class StringReaderScanner {
    public static void main(String[] args) {
        Scanner scn = new Scanner(System.in);
        String message;
        System.out.println("Enter a line of text: ");
        message = scn.nextLine();
        System.out.println("your input is: " +message);

    }
}
```

```
> java StringReaderScanner
    Enter a line of text:
    How are you today?
    your input is: How are you today?
```

- *The **import** statement* allows you to use available methods from other classes/library
- **Tell the system that we will use the Scanner class which is part of java.util library**

- **Define a Scanner object referred by *scn* to invoke methods in Scanner class**
- **The input source is the keyboard**

- **nextLine()** is a method from **Scanner**
- **Reads a whole line as a string**

Interactive Input via Scanner

- Taking input from the terminal is ***good for interaction***
- There are many Scanner ***input*** methods available
  - http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html
  - You can read a line at a time
    - **nextLine():** reads all of the input until end of the line, and returns this whole line as a string
  - Or, you can read one "token" at a time
    - A token is any sequence of non-whitespace characters
    - **next():** reads the next input token as a string and returns this string
      - If the input consists of a series of words separated by whitespaces, each call next() method will return next word

Scanning for primitives

– The Scanner class also contains methods that check for and return primitives directly

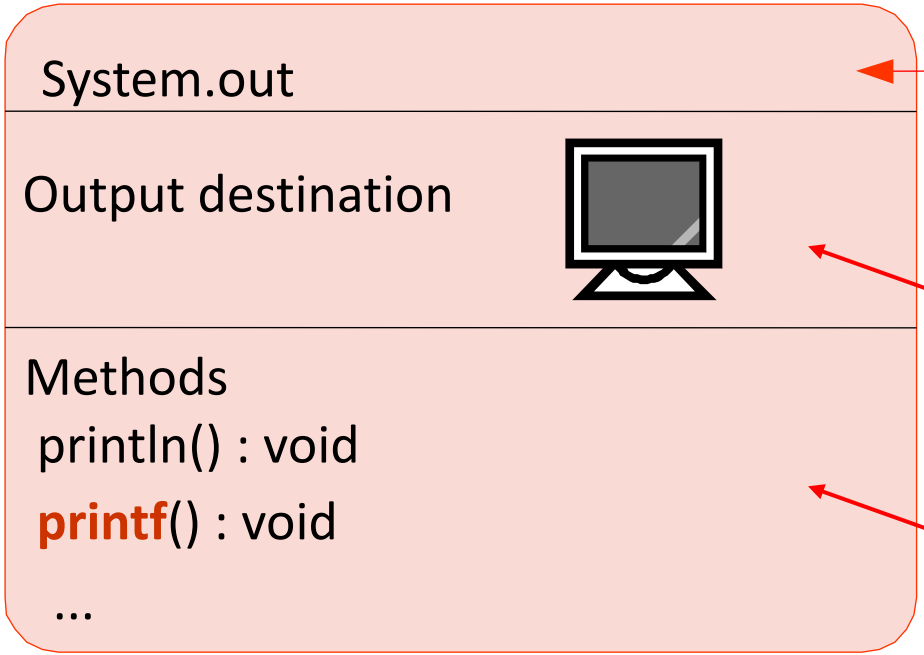
- **Test if you have something to read**
- **Return *true* if there is a token in the type**
  - **hasNextBoolean()**
  - **hasNextByte()**
  - **hasNextShort()**
  - **hasNextInt()**
  - **hasNextLong()**
  - **hasNextFloat()**
  - **hasNextDouble()**

- **You can read in primitives:**

```
boolean b = scan.nextBoolean();
byte by   = scan.nextByte();
short sh  = scan.nextShort();
int i     = scan.nextInt();
long l    = scan.nextLong();
float f   = scan.nextFloat();
double d  = scan.nextDouble();
```

System.out

System.out



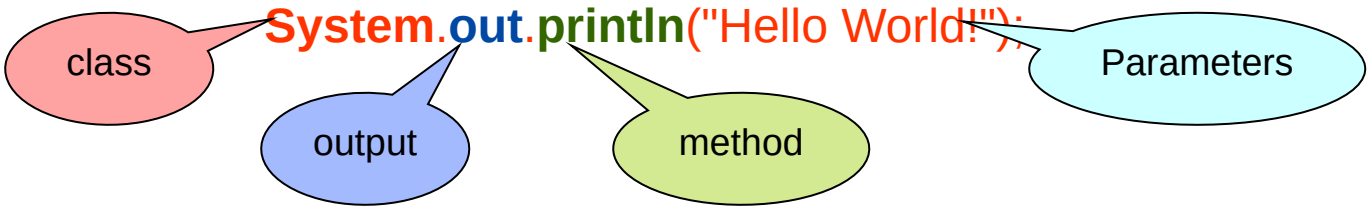
System.out gives access to an output stream

The printing destination is the screen

support different printing methods

Output via *System.out*

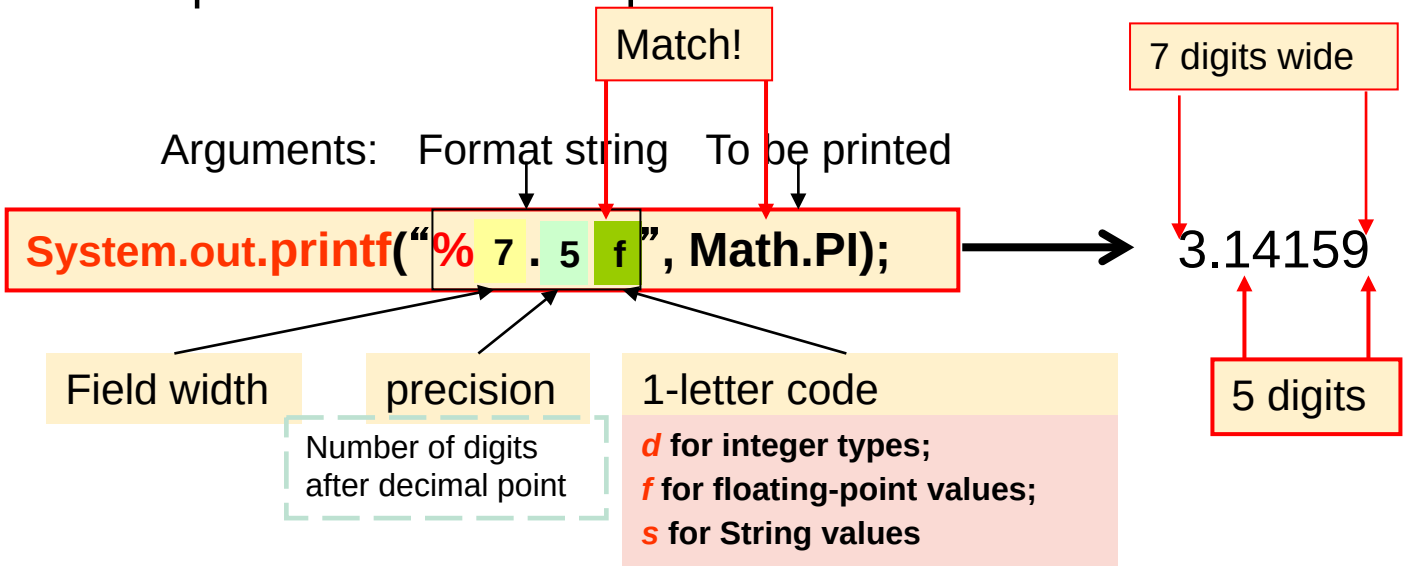
- The ***System*** class contains useful fields and methods to access system resources
- We use ***System.out*** to specify the destination for the output is **system standard output stream (i.e. the monitor screen);**



- The method ***println()*** prints the output and then generates a new line
- The method ***print()*** prints an item without generating a new line

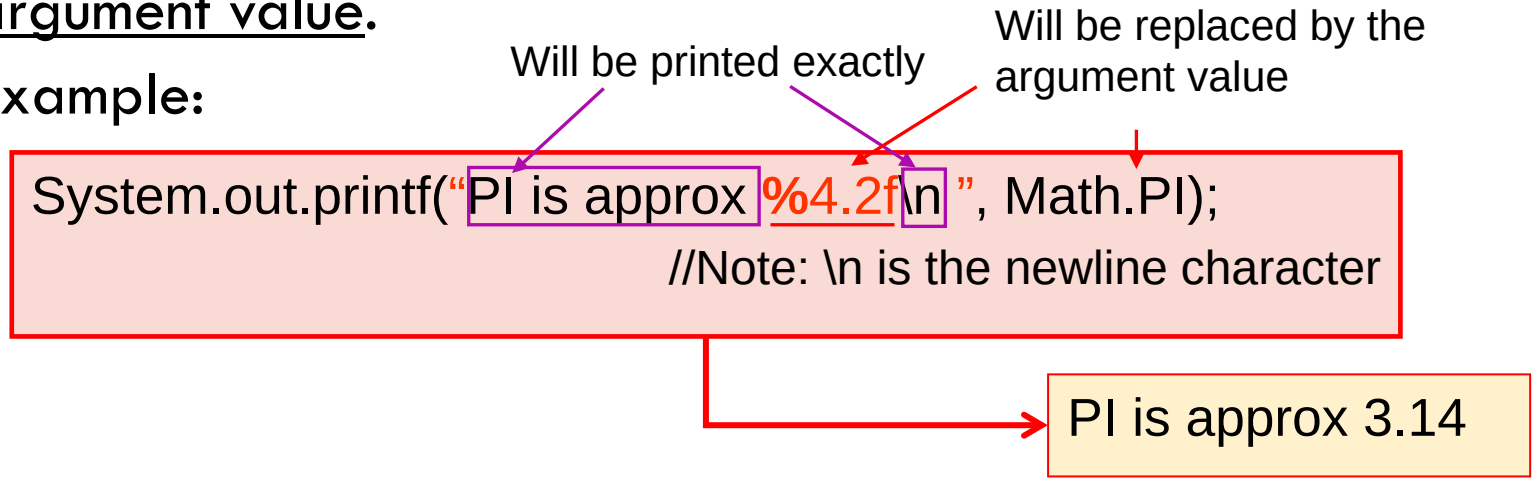
Formatted Output

- The system class also has a method for ***formatted output***
  - ***System.out.printf():*** a flexible method to specify the number of digits and the precision
  - Example of formatted output:



Format String

- The first argument of **printf()** is a string that may contain characters other than a format string.
- Any part of the argument that is not part of format string will be passed to the output, with the format string replaced by the argument value.
- Example:



Multiple Arguments

- The **printf()** function can take multiple arguments. The format string will have a format specifier for each argument value.
  - For example, if we are making a report on payments on a loan, the program may contain the statements as below:

```
System.out.printf("%3s  $%6.2f  $%7.2f     $%5.2f\n",
                  month[i], pay[i], balance[i], interest[i]);
```

```
Jan  $299.00     $9742.66     $41.67
Feb  $299.00     $9484.26     $40.59
Mar  …..
```

User Defined Class

Class Definition

- A class is defined in Java by using the class keyword and specifying a name for it: e.g. public class Customer{ }
- Inside a class it is possible to define:
  - Fields (sets of values, variables, characteristics)
  - Constructors (used to create and initialize new objects)
  - Methods (operations normally defined on the fields/variables)
- Each of these building blocks is qualified by an access modifier/specifier, such as public, private (to be covered next week)
- Syntax: class definition

```
accessSpecifier class ClassName
{
   fields/variables

   constructors

   methods
}
```

- Meaningful class name
- Class name should be noun
- Each word starts with **capital letter**

User defined types

- Java allows programmers to create their own types.
- We can define classes to create new data types by specifying:
  - A set of data values (instance variables)
  - A set of operations that act on these data values (methods)
- In the declaration below
  - i has been declared to be of type int and will remain as type int throughout its life
  - Person is the type and p is the variable.

Example: Customer class

Instance fields

constructors

methods

```
public class Customer {
    private double creditCardBalance;
    private double chequeAccountBalance;
    private String name;

    public Customer(String name) {
        this.name = new String(name);
        this.creditCardBalance = 0;
        this.chequeAccountBalance = 0;
    }

    public Customer(String name, double ccb, double cab) {
        this.name = new String(name);
        this.creditCardBalance = ccb;
        this.chequeAccountBalance = cab;
    }

    public double wealth() {
        return creditCardBalance +
                chequeAccountBalance;
    }

    public boolean inDebt() {
        return (wealth() < 0);
    }

    public void writeCheque(double amount) {
        chequeAccountBalance -= amount;
    }

    public void payWithCreditCard(double amount) {
        creditCardBalance -= amount;
    }

    public void depositChequeAccount(double amount) {
        chequeAccountBalance += amount;
    }
    public void printBalances() {
        System.out.println(name + ": Credit card
balance
        = " + creditCardBalance + ", Cheque
        accountbalance = "+
chequeAccountBalance);
    }
}
```

Fields

Instance Fields

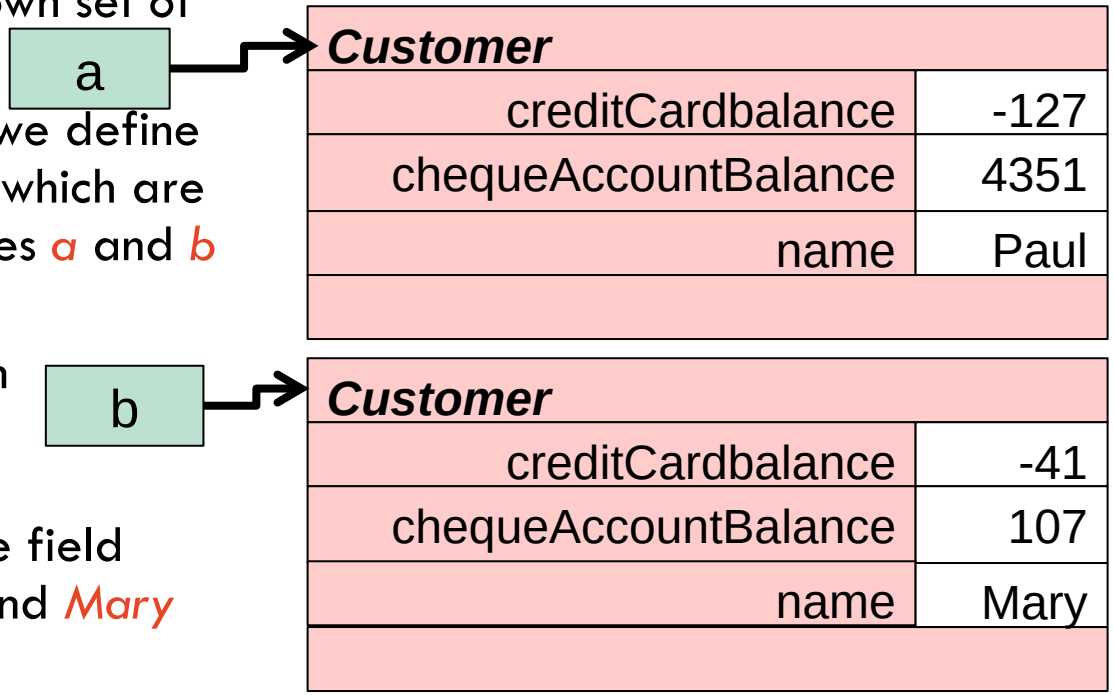
- Syntax: Instance field declaration

  accessSpecifier fieldType fieldName;

- An instance field declaration consists of the following parts:
  - **access specifier** (usually ***private***)
  - type of a field (such as *double*)
  - name of field (such as *balance*, first character is lowercase
- We generally declare all instance fields as ***private***
- Instance fields declared as ***private*** can only be accessed by constructors and methods defined in the same class.

Instance Fields

- Each object of a class has its own set of instance fields.
- For example, in a client class, we define two objects of *Customer* class, which are referred by reference variables *a* and *b* respectively.
- Then each object has their own creditCardBalance field, i.e., *a.creditCardBalance* and *b.creditCardBalance*, and name field with different values as *Paul* and *Mary* respectively.



Constructors

Constructors

- The constructors allow the creation of instances that are properly initialized.

- A constructor is a method that:
  - has the same name as the name of the class to which it belongs
  - has no specification for the return value, since it returns nothing.

Default constructors

- The constructor with zero parameters is called the *default constructor*.
- Java provides a default constructor for the classes.
- This default constructor is only available when no constructors are defined in the class.

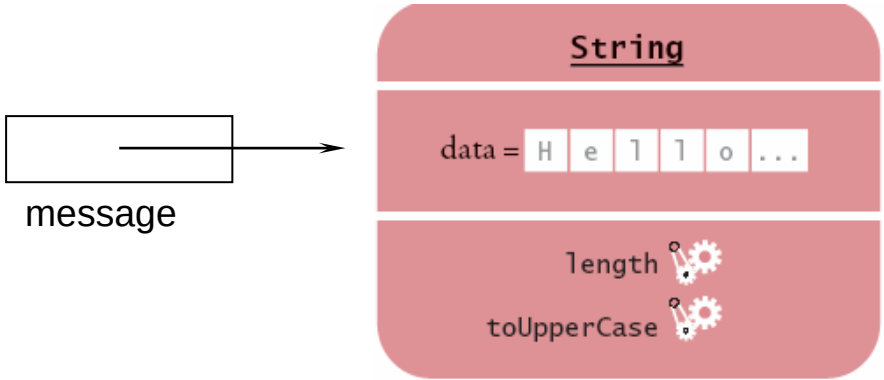
Constructors: Creating Objects

- The class constructor is called by using ***new***:

  ***new* class-name(parameter-list);**

- `String()` is a String constructor which is used to allocate memory and to initialize ***instance variables***
  - Example:

    String message = **new String**("Hello, World!");



Multiple constructors

- It is possible to define more than one constructor for a single class, only if they have different number of arguments or different types for the arguments.

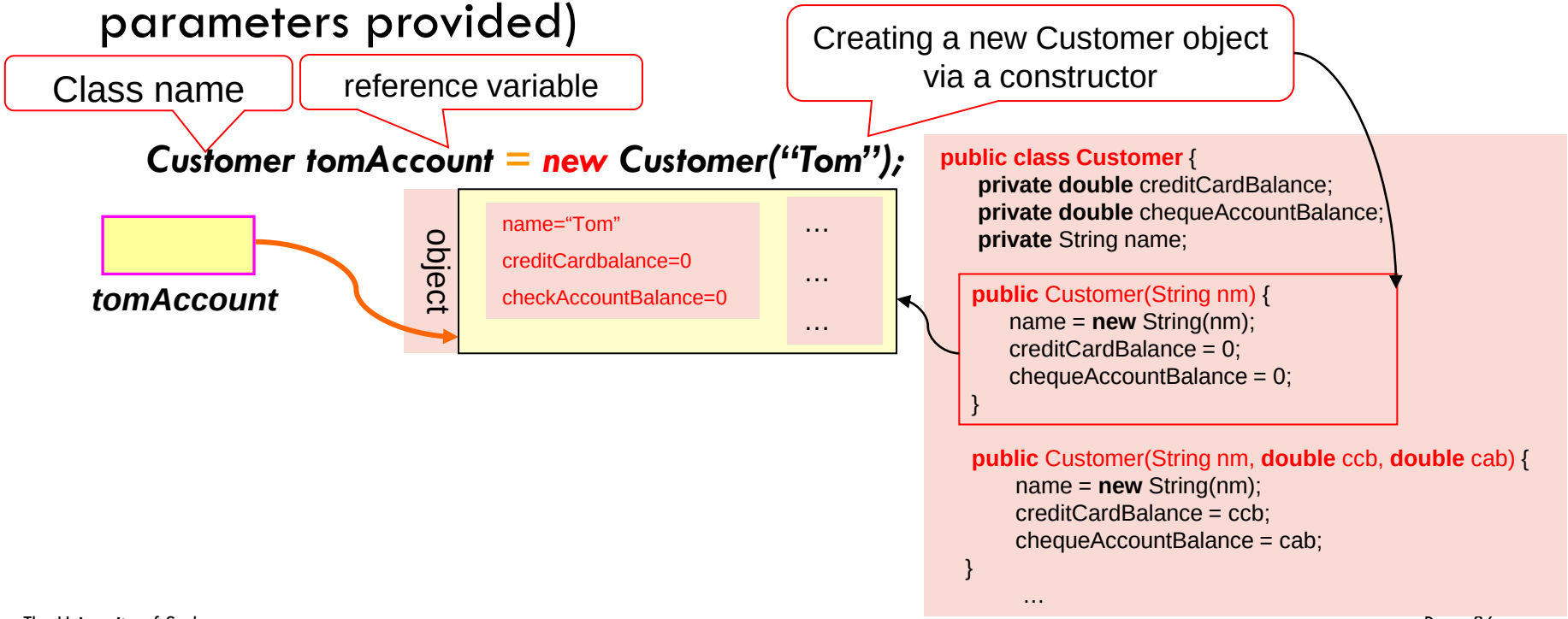
```
public Customer(String nm) {
            name = new String(nm);
            creditCardBalance = 0;
            chequeAccountBalance = 0;
    }
public Customer(String nm, double ccb, double cab)    {
            name = new String(nm);
            creditCardBalance = ccb;
            chequeAccountBalance = cab;
    }
```

Constructors -- Customer objects can be constructed in 2 ways:

1. Initializing with just a name. The balances are set to 0.
2. Initializing with a name and balances for both accounts.

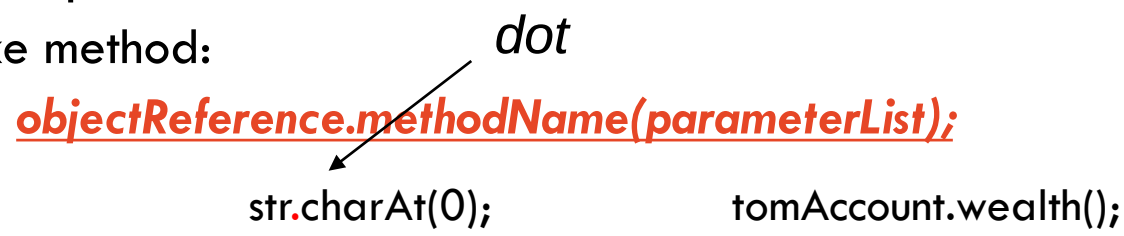
Constructors: Creating Objects

- The compiler calls the appropriate constructor (based on the parameters provided)



Accessing Objects

- Once the object has been instantiated, we can use **dot operator**
  - to access its data
  - to invoke its well-defined methods to manipulate the object
    - A method is a sequence of instructions that can access the data of the object
    - A method invocation can be thought as asking an object to perform a service
- Invoke method:



Example

```
public class PrintCharacters {
        public static void main(String[] args) {
                System.out.print("enter a word: ");
                String word = new String(args[0]);
                System.out.println(word); //print the word
                for (int i = 0; i < word.length(); i++) {
                        char current = word.charAt(i);
                                        //the current letter in the word to be handled
                        System.out.println("letter " + i +" is: "+ current);
                        // print the letter
                }
        }
}
```

String Creation

Method Invocation

How would you change this to:

- Convert the word to upper/lower case before printing the letters?
- How to separate the dollar sign $ from your wage?
- How to remove any leading zeroes in a given phone number?

Methods

Methods

- A method is used to implement the messages that an instance (or a class) can receive.
- Methods are important because they allow us to
  - Clearly separate tasks within a program and make it easier to understand, to debug and to maintain
  - **Reuse code**
- It is called by using the dot notation.
  - Examples
    - Built-in methods:
      - Math.random(), Math.abs(), Integer.parseInt()
    - The **main()** method is the start point of running a program.

Method Definition Syntax

specifiers are words to set characteristics of the method. E.g.,"**public**", "**static**"

Specify the type of value returned from the method. "**void**" indicates that the no value be returned from the method.

Represent information passed to the method from method invoker/user. Parameter-list can be empty, or consist of one or more parameter declarations in the format of **type parameter-name** for each parameter, separated by **commas**.

specifiers *return-type* *methodName* *(parameter-list)*

Method body (black box)

statements;

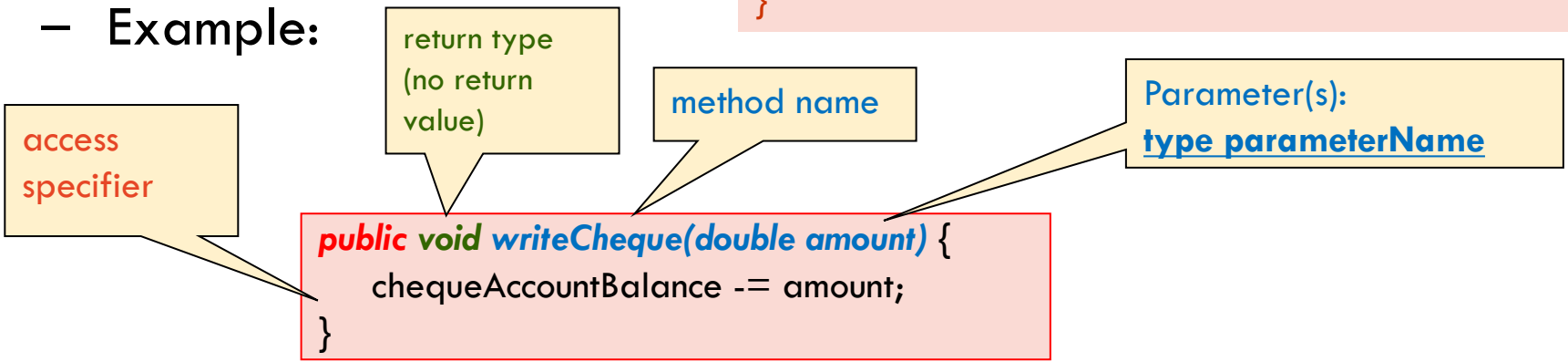
Interaction interface

Method Definition

- Method Definition Format

```
specifier returnType methodName(parameterList)
{
   method body //Java statements to perform certain tasks
}
```

- Example:



- The modifier *public* indicates that the method can be called anywhere in a program even from outside the class where this method is defined.

Method Definition

```
public void depositChequeAccount (double amount) {
        chequeAccountBalance += amount;
}
```

- This method is an instance method
- Instance methods are invoked from an object
- The instance methods can be used only after the object is created
- Example:

```
//create an object in a client class of Customer class
Customer harryAcc = new Customer("Harry", 1000, 50);
//invoke the deposit method from the available/created object
harryAcc.depositChequeAccount(5000);
```

Method Parameters

- **Parameters** are a vehicle for passing information to a method.
- **Scope of parameter variables:** You can use parameter variables anywhere in the method body.
- **Arguments (or actual parameters):** the actual values passed to the method when it is called.
  - The actual argument can be specified by an expression.
- The parameter variable is initialized with the actual argument value provided by the calling code.

Example: Method Parameters

```
public class NewtonExample {

    public static double sqrt( double   c)
    {
        if (c < 0) return Double.NaN;
        double EPS = 1E-15;
        double t = c;
        while (Math.abs(t - c/t) > EPS*t)
                t = (c/t + t) / 2.0;
        return t;
          }

    public static void main(String[ ] args)
    {
            double x = sqrt(25.0);
            System.out.println(x);

    }
}
```

c=25

This method has one parameter

Start point of program execution

Argument (the actual value passed to the method when it is invoked)

Key Concept: When a method is called, the value of actual parameters are copied into the formal parameters

Primitive Parameters

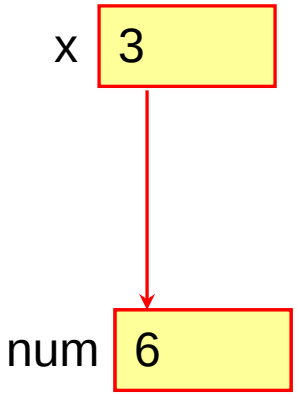
– Parameters of primitive data types

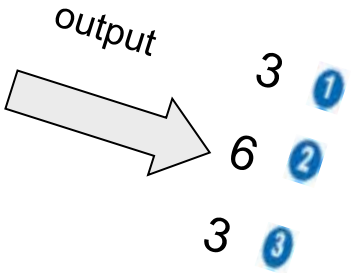
```
public static void main (String [ ] args)
{
    int x=3;
    System.out.println(x);
    doubleNum(x);
    System.out.println(x);
}

public static void doubleNum (int num)
{
    num=num*2;
    System.out.println(num);
}
```

variable *x* in *main()* is unchanged

x=3 was copied to parameter *num* *(i.e., num=3)*

*num=num*2 (i.e., num=6 and printed)*





Changes to the value of a primitive parameter inside a method will NOT change actual argument of primitive types when the method exits

Arrays as Parameters

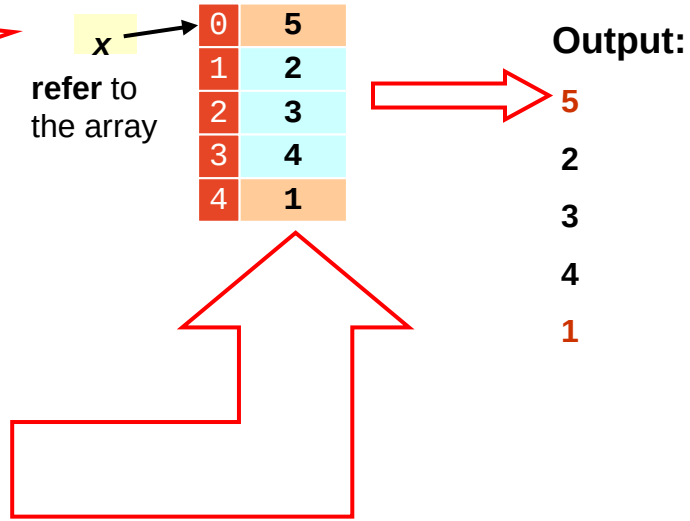
- When a method takes an array as parameter, it implements a function that **operates on an arbitrary number of values of the same type.**
- For example, the following method computes the mean value of an array of *double* values

```
public double mean (double[ ] a)
{
        double sum=0.0;
        for (int i=0;i<a.length;i++)
                sum=sum+a[i];
        return sum/a.lenght;
}
```

Arrays as Parameters

- Methods that takes an array as a parameter may **change the values in the array**

```
public static void main (String [ ] args)
{
  int [ ] x = {1, 2, 3, 4, 5};
  for(int i=0; i<x.length; i++)
        System.out.println(x[i]);
  exch(x, 0, x.length-1);
  for(int i=0; i<x.length; i++)
        System.out.println(x[i]);
}

public static void exch(int[ ] a, int i, int j)
{           int t=a[i];

            a[i]=a[j];

            a[j]=t;

}
```



The parameter variable *a* in *exch()* is a reference to the array. When you pass an array argument to a an array parameter, the parameter and the argument share the same array.

Call by Value and Call by Reference

Parameter is a primitive type

- Value of the argument is copied to the method's parameter
- The changes to the parameter value in the method will **NOT** change the original value of the argument
- **Call by value**

Parameter is a reference

- Reference (address) of the argument is copied to the method's reference parameter
- The changes of the reference parameter in the method will **AFFECT** the argument
- **Call by "reference"**

The *return* Statement

- The value returned from a method must match (be consistent with) the return type specified in the method definition/signature/header.
- return (a reserve word) is used to bring back an output value
- An explicit return is not necessary in a void method
  - E.g., main() method that you have used before does not have a return statement;
  - However, return can be used to cause immediate termination of the method

The *return* Statement

- ***return*** used to cause **immediate termination** of the method

```
public void stopMethod()
{
    for (int i =1; i<10; i++){
        if (i==5) return;
        System.out.println(i);
    }
}
```

Will this print statement be performed when *i==5*?

Will this print statement be performed when *i==5*?

- Used for **returning a value**

Same type

```
public double sqrt(double c)
{
    if (c < 0) return Double.NaN;
    double EPS = 1E-15;
    double t = c;
    while (Math.abs(t - c/t) > EPS*t)
        t = (c/t + t) / 2.0;
    return t;
}
```

Multiple Methods: Overloading

- In a class, we can define multiple methods with same name but with different parameter lists
- For example: we often want to define the same operation for values of different types

```
public class MultiMethods {

    public static int abs (int x){
        if (x < 0) return –x;
        else return x;
    }

    public static double abs (double x){
        if (x < 0.0) return –x;
        else return x;
    }

  public static void main(String[]args){
      System.out.println(abs(-1.6));
  }
}
```

Two methods with identical name but different parameter types

Example: Customer class

Instance fields

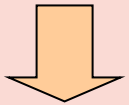
constructors

methods

```
public class Customer {
    private double creditCardBalance;
    private double chequeAccountBalance;
    private String name;

    public Customer(String nm) {
        name = new String(nm);
                creditCardBalance = 0;
        chequeAccountBalance = 0;
    }

    public Customer(String nm, double ccb, double cab) {
        name = new String(nm);
        creditCardBalance = ccb;
        chequeAccountBalance = cab;
    }

    public double wealth() {
        return creditCardBalance +
                    chequeAccountBalance;
    }

    public boolean inDebt() {
        return (wealth() < 0);
    }

    public void writeCheque(double amount) {
            chequeAccountBalance -= amount;
    }

    public void payWithCreditCard(double amount) {
            creditCardBalance -= amount;
    }

    public void depositChequeAccount(double amount) {
            chequeAccountBalance += amount;
    }
    public void printBalances() {
            System.out.println(name + ": Credit card
    balance
             = " + creditCardBalance + ", Cheque
             accountbalance = "+
    chequeAccountBalance);
    }
}
```

Example: Customer class

```
// a tester class used to test Customer class
public class CustomerTester {
      public static void main(String[] args) {
            Customer harrysChecking = new Customer(“Harry”);
            Customer xyAccount = new Customer("xiuying", 20, 50);

            harrysChecking.depositChequeAccount(2000);
            harrysChecking.writeCheque(500);
            harrysChecking.writeCheque(500);

            xyAccount.depositChequeAccount(2000);
            xyAccount.depositChequeAccount (1000);
            xyAccount.writeCheque(500);
            xyAccount.printBalances();
            harrysChecking.printBalances();
           System.out.println("Harry's wealth: " +harrysChecking.wealth());

   }

}
```



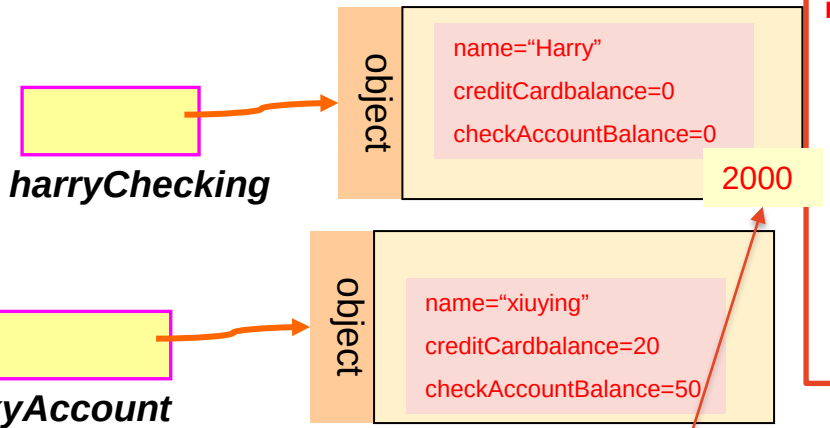
```
xiuying: Credit card balance = 20; Cheque account balance = 2550
Harry: Credit card balance = 0; Cheque account balance = 1000
Harry's wealth: 1000
```

Question: Is the statement below correct? Try it on your computer

```
System.out.println(xyAccount.creditCardBalance);
```

Example: Customer class

```
// a tester class used to test Customer class
public class CustomerTester {
    public static void main(String[] args) {
        Customer harrysChecking = new Customer("Harry");
        Customer xyAccount = new Customer("xiuying", 20, 50);
    …
```




```
public class Customer {
    …
    public Customer(String nm) {
        name = new String(nm);
        creditCardBalance = 0;
        chequeAccountBalance = 0;
    }
    public Customer(String nm, double ccb, double cab) {
        name = new String(nm);
        creditCardBalance = ccb;
        chequeAccountBalance = cab;
    } .
```

```
harrysChecking.depositChequeAccount(2000);
harrysChecking.writeCheque(500);
```

```
public void writeCheque(double amount) {
        chequeAccountBalance -= amount;
}

public void depositChequeAccount(double amount) {
        chequeAccountBalance += amount;
}
```

Questions?