

CMPS 251



Read Chapters 3 & 6

Basic Object-Oriented Programming in Java

Dr. Abdelkarim Erradi

CSE@QU

Outline

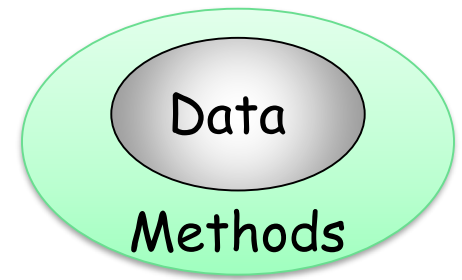
- Classes and Objects
- Attributes with Getters and Setters
- Methods
- Static Methods / Attributes
- Constructors
- Value Types vs. Reference Types
- JavaDoc Comments

Classes and Objects

Classes

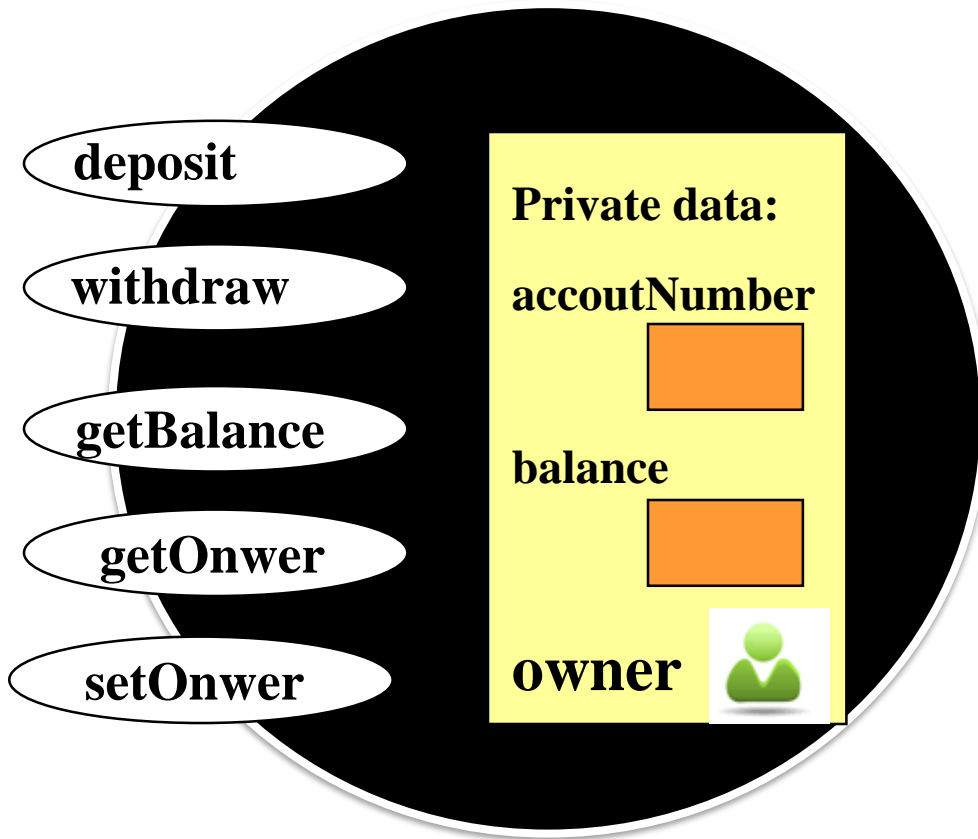
- A class is a **programmer-defined data type** and **objects are variables of that type**
 - Classes allow us to create new data types that are well suited to an application
 - A class contains private **attributes** and **methods**
- An object is an **instance** of a class
e.g., `Student quStudent = new Student;`
This declares quStudent object of type Student. The object is then created using the **new** keyword

Encapsulation



- **Encapsulation** = an object **combines attributes and methods into a single unit**
 - Hiding implementation from clients
 - Clients access the object via its **public methods (public methods = *interface*)**
 - The data is *hidden*, so it is safe from any accidental alteration
 - Get and Set methods are used to read/write the object's attributes

BankAccount Example



BankAccount contains **attributes**
and **methods**

An Object has:

- **Attributes** – information about the object
- **Methods** – functions the object can perform
- **Relationships** with other objects
 - e.g., A **BankAccount** has an **Owner**
 - A relationship is defined also as an attribute

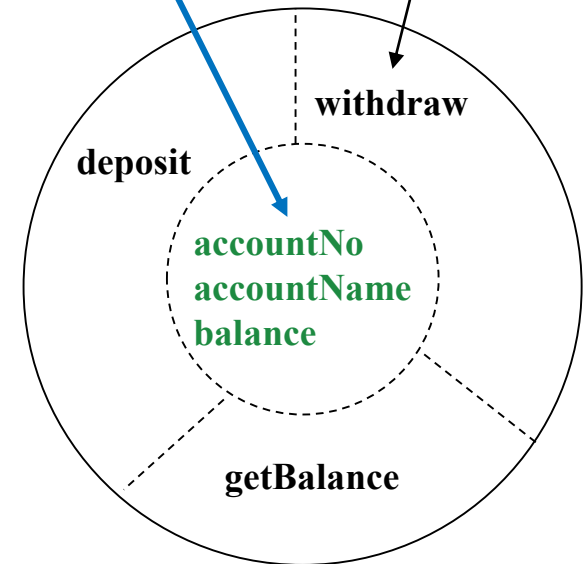
Encapsulation - Example

```
public class Account {  
    private int accountNo;  
    private String accountName;  
    private double balance;  
  
    public void deposit(double amount) {  
        balance += amount;  
    }  
    public void withdraw(double amount) {  
        balance -= amount;  
    }  
    public double getBalance() {  
        return balance;  
    }  
    .....  
}
```

private data

Attributes

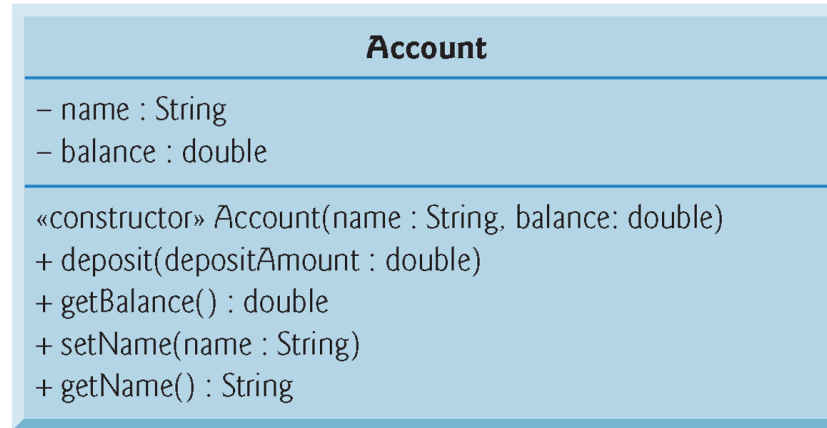
Methods



Bank Account Object

public methods

UML class diagram for class Account



- Unified Modeling Language (UML) class diagram
 - A rectangle with three compartments
 - Top compartment contains the **name of the class**
 - Middle compartment contains the **class's attributes**
 - (-) in front of an attribute indicates it is private
 - Bottom compartment contains the **class's functions**
 - (+) in front of a method indicates it is public

Using a class

- A class is a programmer-defined type
 - Can be used to create objects i.e., variables of the class type
 - A class can be viewed as a *factory* for objects
 - To use a class you must create an object from a class (this is called **instantiation**)
 - To drive a car you must manufacture the car based on its design!
- A class can be used as the type of an attribute or local variable or as the return type of a method
- **Dot operator (.)**
 - Used to **access an object's attributes and call its methods**
 - **Call causes the method to perform its task.**
 - e.g. : `myGradeBook.displayMessage("Welcome to CMPS251 Computer Programming")`
 - Calls the method `displayMessage`

Instantiation

- Instantiation = Object creation with ***new*** keyword
- **Memory is allocated for the object's** attributes as defined in the class
- Initialization of the object attributes as specified through a ***constructor***
 - ***constructor*** a special method invoked when objects are created
 - Constructors are discuss further in these slides

Java Naming Conventions

- Start classes with uppercase letters

```
/** Short description of the class */  
public class MyClass {  
    ...  
}
```

Use JavaDoc-style
comments

- Start other things with lowercase letters: attributes, local variables, methods, method parameters

```
public class MyClass {  
    private String firstName, lastName;  
    public String getFullName() {  
        String name = firstName + " " + lastName;  
        return name;  
    }  
}
```

Attributes

Attributes

- **Attributes** = data that is stored inside an object. Also called 'Instance variables', 'data members' or object state

- Syntax

```
public class MyClass {  
    private SomeType attribute1;  
    private SomeType attribute2;  
    //ToDo: provide getters and setters  
}
```

It is conventional to
make all attributes
private

- Motivation

- Lets an object have data values (i.e., a state)
 - In OOP, objects have 2 characteristics: **state** and **behavior**. The attributes provide the state.

Attributes

- Attributes = Local variables in a class definition
 - Exist throughout the life of the object
 - **Each object of a class maintains its own copy of attributes**
 - e.g., different accounts can have different balances
- Access-specifier **private** used for Data hiding
 - Makes an attribute or a method accessible only to methods of the class
- An attempt to access a **private** attribute outside a class is a compilation error

Accessor methods

- Ideas
 - Attributes should always be private
 - And made accessible to outside world with **get** and **set** methods (also know as Accessor methods)
- Syntax

```
public class MyClass {  
    private String firstName;  
    public String getFirstName()  
        { return(firstName); }  
    public void setFirstName(String firstName)  
        { this.firstName = firstName; }  
}
```
- Motivation
 - Allow data validation before assigning values to the object attributes
 - Limits accidental changes

Accessor methods

get Methods and *set* Methods

- Best practice is to provide a **get** method to **read** an attribute and a **set** method to **write** to an attribute
 - Data is protected from the client. Get and set methods are used rather than directly accessing the attributes
 - Using *set* and *get* functions allows the programmer to control how clients access private data + **allow data validation:**
 - Can return errors indicating that attempts were made to assign invalid data
 - *set* and *get* methods should also be used even inside the class

Examples of Validation

```
class Date {  
    int day, month;  
    // setDay assigns its argument to the private member day.  
    public void setDay(int day)  
    {  
        if (day >= 1 && day <= 31)  
            this.day = day;  
        else  
            throw new Exception("The day must me between 1 and 31");  
    }  
    // setMonth assigns its argument to the private member month.  
    public void setMonth(int month)  
    {  
        if (month >= 1 && month <= 12)  
            this.month = month;  
        else  
            throw new Exception("The month must me between 1 and 12");  
    }  
}
```

Exception will be
discussed later

! An attempt to access a `private` member outside a class is a syntax error

Not all attributes need Getters and Setters

- Some attributes might not need both getters and setters. They should be provided only when appropriate
 - It is common to have fields that can be set at instantiation, but never changed again (immutable field).

```
public class Student {  
    private final String studentId;  
    public Student(String studentId) {  
        this.studentId = studentId;  
    }  
    public String getStudentId() { return(studentId); }  
    // No setStudentId method  
}
```

final = value of the attribute cannot be changes once initialized

Generating Getters and Setters

- Eclipse will automatically generate getters/setters from instance variables
 - R-click anywhere in code
 - Choose Source → Generate Getters and Setters
 - However, if you later click on instance variable and do Refactor → Rename, Eclipse will not automatically rename the accessor methods

Methods

Overview

- Definition

- Functions that are defined inside a class. Also called “member functions”.

- Syntax

Use void if the method returns nothing.

```
public ReturnType methodName(Type1 param1, Type2 param2, ...) {  
    ...  
    return somethingOfReturnType;  
}
```

If the method is called only by other methods in the same class, make it private.

- Motivation

- Lets an object calculate values or do computations, usually based on its current state (i.e. attributes)
 - In OOP, objects have **state** and **behavior**. The **methods** provide the behavior

Calling Methods

- The usual way that you call a method is by doing the following:

```
variableName.methodName(argumentsToMethod);
```

- For example, the built-in `String` class has a method called `toUpperCase` to convert to uppercase
 - This method doesn't take any parameters, so you just put empty parentheses after the method name.

```
String s1 = "Hello";
```

```
String s2 = s1.toUpperCase(); // s2 is now "HELLO"
```

Method Visibility

- public/private distinction
 - A declaration of **private** means that “outside” methods cannot call it – only methods within the same class can
 - Attempting to call a private method outside the class would result in an error at compile time
 - Only use **public** for methods that *your class will make available to users*
 - You are **free to change or eliminate private** methods without telling users of your class
- Attributes should be private by convention

Methods Overloading

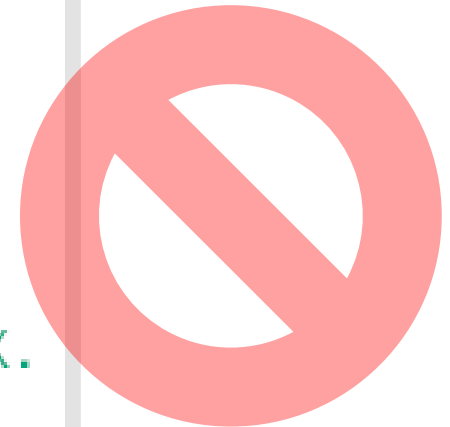
- Idea
 - Classes can have more than one method (or constructor) with **the same name**
 - The methods (or constructors) have to differ from each other by **having different number or types of parameters** (or both), so that Java can always tell which one you mean
 - A method's name and number and type of parameters is called the ***signature***
- Syntax

```
public class MyClass {  
    public double getRandomNum() { ...};  
    public double getRandomNum(double range) { ... }  
}
```
- Motivation
 - Lets you have the same name for methods doing similar operations (***ease learning and understanding your program***)
 - Overloaded constructors let you ***build instances in different ways***

Overloading and Return Type

- You must not overload a method where the only difference is the type of value returned

```
/**  
 Returns the weight of the pet.  
 */  
public double getWeight()  
  
/**  
 Returns '+' if overweight, '-' if  
 underweight, and '*' if weight is OK.  
 */  
public char getWeight()
```



Static Methods / Attributes

Compare these two alternatives ...

```
// This makes little sense  
Math math = new Math();  
float answer = math.sin(45);
```

```
// This would make more sense  
float answer = Math.sin(45);
```

Static Methods are often used for Helper Class such as Math Class

Static Methods

- A static method belongs to the class rather than the object of the class.
 - Also known as “class methods” (vs. “instance methods”)
- A static method can only access the static attributes.
- A static method can be called without the need for creating an instance of the class.
 - You call a static method through the class name

ClassName.functionName(arguments);

- For example, the **Math** class has a static method called **cos**
 - You can call **Math.cos(3.5)** without creating an object of the **Math** class
- E.g., the `main` method is a static method so the system can call it without first creating an object

Static Methods vs. Instance Methods

- In a class **Car** you might have a method
`static double convertMpgToKpl(double mpg)`
 - which would be static, because one might want to call it even without creating a car object
(mpg : miles per gallon. Kpl : Km per litre)
 - Better design is to even moved it to a utility class
- But

`void setMileage(double mpg)`

can't be static since it's inconceivable to call the method before any Car has been constructed.

When to use Static Methods

- Define static methods in the following scenarios only:
 - If you are writing **utility or helper classes** e.g., Math class
 - e.g., `int answer = Math.sin(45);`
 - Classes for which **only one instance is needed for the whole application**
e.g. `System.out` is a static attribute as only 1 display output object is needed for the whole application to output to the screen
 - If the method is not using any instance variable and doesn't need any object to be initialized for it to be called
 - Methods such as **sorts or comparisons** that operate on multiple objects of a class and are not tied to any particular instance. E.g.,
`Car getMoreEfficient(Car car1, Car car2)`
 - Wrapper classes –see next slide–

Wrapper Classes

- Java provides *wrapper classes* for each primitive type
- Allow programmer to have an object that corresponds to value of primitive type
- Contain useful predefined constants and methods
- Wrapper classes have no default constructor and no **set** methods
 - Programmer must specify an initializing value when creating new object

Wrapper Class Example: Static methods in **Character** class

Name	Description	Argument Type	Return Type	Examples	Return Value
toUpperCase	Convert to uppercase	char	char	Character.toUpperCase('a') Character.toUpperCase('A')	'A' 'A'
toLowerCase	Convert to lowercase	char	char	Character.toLowerCase('a') Character.toLowerCase('A')	'a' 'a'
isUpperCase	Test for uppercase	char	boolean	Character.isUpperCase('A') Character.isUpperCase('a')	true false
isLowerCase	Test for lowercase	char	boolean	Character.isLowerCase('A') Character.isLowerCase('a')	false true
isLetter	Test for a letter	char	boolean	Character.isLetter('A') Character.isLetter('%')	true false
isDigit	Test for a digit	char	boolean	Character.isDigit('5') Character.isDigit('A')	true false
isWhitespace	Test for whitespace	char	boolean	Character.isWhitespace(' ') Character.isWhitespace('A')	true false

Whitespace characters are those that print as white space, such as the blank, the tab character ('\t'), and the line-break character ('\n').

Static Attributes

- Static attributes also called *class attributes* = NOT associated with any instance of the class.
 - Contrast with *instance attributes*
- Static attributes are **shared** by all objects
 - Only one instance of the attribute exists
 - It can be accessed by all instances of the class
- Attributes declared using **static final** are considered constants and their value **cannot** be changed (e.g., Math.PI)



Value Types vs. Reference Types

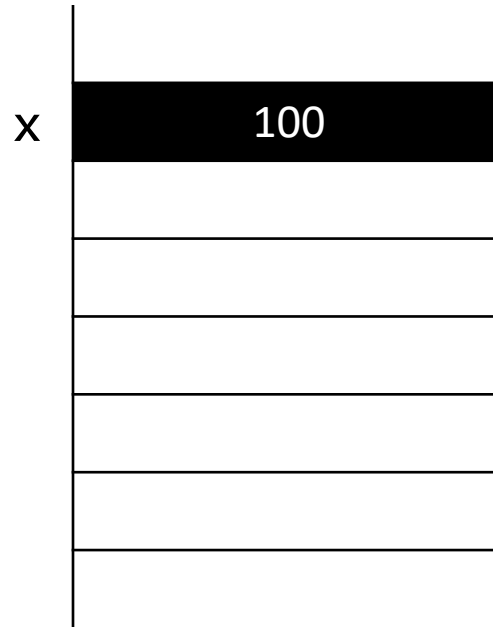
There are two types of variables

1 Value:
Stores the actual value

2 Reference:
Stores a reference to the actual value

Value types store values

```
int x = 100;
```

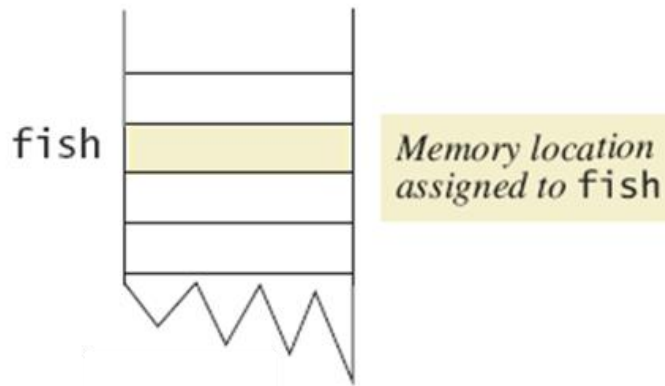


The value (also called primitive) types in Java are boolean, byte, char, short, int, long, float and double.

Using **new** ClassName constructs the object and **returns** a reference

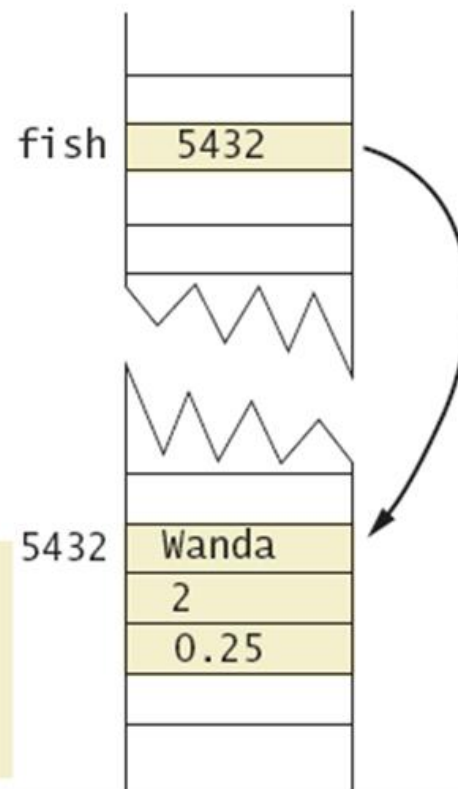
```
Pet fish;
```

Assigns a memory location to fish



```
fish = new Pet();
```

Assigns a chunk of memory for an object of the class Pet—that is, memory for a name, an age, and a weight—and places the address of this memory chunk in the memory location assigned to fish



Objects and References

- Once a class is defined, you can declare variables (object reference) of that type

```
Ship ship1, ship2;  
Point startPoint;  
Color blue;
```

- Object references are initially **null**
 - The **null** is a special value in Java indicating that the object is NOT created yet
- The **new** operator is required to create the object

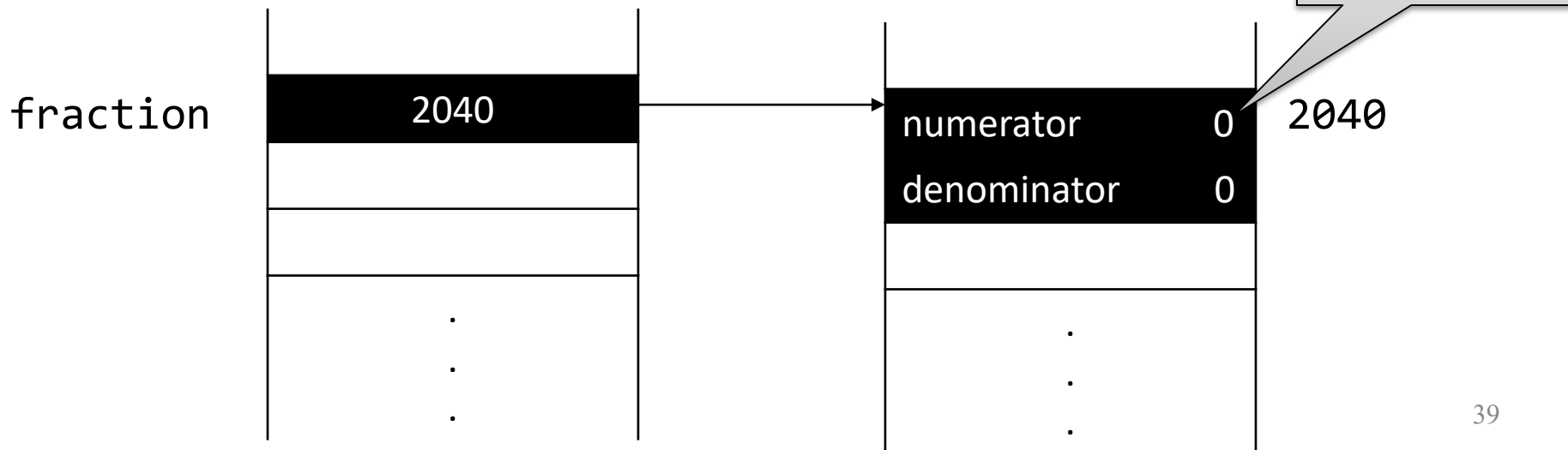
```
ClassName variableName = new ClassName();
```

Reference types store references

Consider the Fraction class:

Fraction	
-	numerator: int
-	denominator: int
+	getNumerator(): int
+	setNumerator(int n): void
+	getDenominator(): int
+	setDenominator(int d): void
+	Fraction(int n, int d);

```
Fraction fraction = new Fraction();
```



Lets compare value types

```
int x = 100;
```

x 100

```
int y = 200;
```

y 200

```
x == y ?
```

false

```
x = y;
```

x 200

y 200

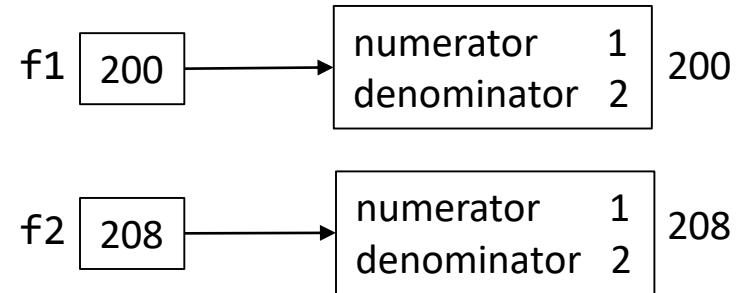
```
x == y ?
```

true

Now lets compare reference types

```
Fraction f1 = new Fraction(1, 2);
```

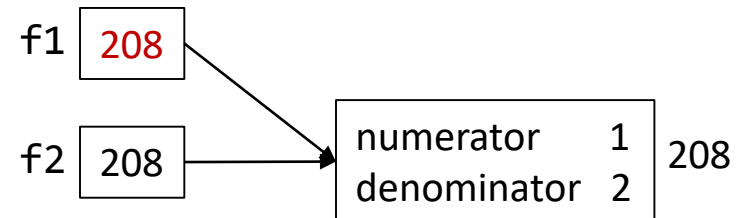
```
Fraction f2 = new Fraction(1, 2);
```



```
f1 == f2 ?
```

false

```
f1 = f2;
```



```
f1 == f2 ?
```

true

The solution to the == problem?

Define an equals method

```
public class Fraction {  
    ...  
    public boolean equals(Fraction f) {  
        return this.numerator == f.numerator &&  
            this.denominator == f.denominator;  
    }  
}
```

Two fractions are equal if both their numerator and denominator values are the same.

Code Deconstructed

<equals method>

```
Fraction f1 = new Fraction(1, 2);
```

```
Fraction f2 = new Fraction(1, 2);
```

```
if (f1 == f2)
    System.out.println("Both variables refer to the same object");
else
    System.out.println("Each variable refers to a different object");
```

```
if ( f1.equals(f2) )
    System.out.println("Both objects have the same value");
else
    System.out.println("Each object has a different value");
```

Constructors

Overview

- Definition
 - Constructor is a special initialization method called when an object is created with **new**.
- Syntax
 - Constructor has the same name as the class and has no return type (not even void).

```
public class MyClass {  
    public MyClass(Type paramName, ... ) { ... }  
}
```
- Motivation
 - Lets you create an instance of the class and at the same time **initialize** the object attributes
 - Lets you **enforce the initialization of some object attributes** at the time of the object creation
 - Lets you run some custom initialization logic when the class is instantiated (e.g., open a file)

Initializing Objects with Constructors

- **Constructors** = **'Methods'** used to initialize an object's attributes when it is created
 - **Call made implicitly** when the object is instantiated
 - There is no explicit way to call the constructor
 - A constructor has the same name as the class and has no return type - Not even `void`
- A class can have zero, one or more different constructors
 - We distinguish between constructors using different number and types of parameters
- Default constructor has no parameters
 - Java will define this automatically if the class does not have any constructors
 - If you do define a constructor, Java will not automatically define a default constructor

Constructor Example

- Constructor = special method that handles the object initialization
- A constructor **is used to initialize the objects during object construction.**

- Example:

```
Account saraAcct = new  
Account(123, "Sara",  
          "Saving");
```

```
public class Account {  
    private int id;  
    private String name;  
    private String type;  
    private double balance;
```

Constructor

```
    Account (int id, String name, String type) {  
        this.id = id;  
        this.name = name;  
        this.type = type;  
        balance = 0;  
    }  
  
    ...  
}
```

The `this` Variable

- The `this` object reference can be used inside any non-static method to **refer to the current object**
 - `this` is used to **refer to the object** from inside its class definition
 - The keyword `this` stands for the receiving object
- `this` is commonly used to **resolve name conflicts**
 - Using **`this`** permits the use of attributes in methods that have local variables / parameters with the same name

```
public void setName(String name) {  
    this.name = name;  
}
```

This is an attribute. To avoid confusion we add `this.` in-front of it

This is a parameter

JavaDoc

Comments and JavaDoc

- Java supports 3 types of comments
 - `//` Comment to end of line.
 - `/*` Block comment containing multiple lines. `*/`
 - `/**` A JavaDoc comment placed before class definition and methods. Text may contain HTML tags, hyperlinks, and JavaDoc tags. `*/`

- JavaDoc is used to generate on-line documentation

```
javadoc Foo.java Bar.java -d outputdir
```

```
javadoc *.java -d docs
```

- More information about JavaDoc available @

<https://docs.oracle.com/en/java/javase/12/tools/javadoc.html>

Useful Javadoc Tags

- **@author**

- Specifies the author of the document
- Must use `javadoc -author ...` to generate in output

```
/** Description of some class ...
 *
 *      @author <a href="author@coding.com">
 *              Java Programmer</a>
 */
```

- **@version**

- Version number of the document
- Must use `javadoc -version ...` to generate in output

- **@param**

- Documents a method argument

- **@return**

- Documents the return type of a method

JavaDoc comment example

```
/** An example to demonstrate Javadoc comment
 *  @author <a href="erradi@coding.com"> Erradi </a>
 *  A method that determines the quotient of two integers.
 *  @param x
 *      The number that is going to be divided
 *  @param y
 *      The number that x is going to be divided by
 *  @return
 *      The quotient that is the result of x/y.
 *  @throws ArithmeticException
 *      Indicates that y is equal to zero.
 */
public static int divide(int x, int y) throws
ArithmeticException
```

JavaDoc Example

*javadoc *.java*

Package

Class

Use

Tree

Index

Help

Prev Class

Next Class

Frames

No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

QuBank

Class Account

java.lang.Object
 QuBank.Account

public class **Account**
extends java.lang.Object

Account example to demonstrate OOP in Java.

Author:
 Java Programmer

Constructor Summary

Constructors

Constructor and Description
<code>Account(int accountNo, java.lang.String accountName)</code>
<code>Account(int accountNo, java.lang.String accountName, double balance)</code>
Build account with specified parameters.

53

Programming Conventions and Best Practices

- Make all attributes private and provide getters and setters
- Class names start with upper case
- Method names and variable names start with lower case
- Choose **meaningful names** for classes, methods and variables
 - makes programs more readable and understandable
- Use JavaDoc-style comments to generate useful documentation
- Indent nested blocks consistently to communicate the structure of your program
 - **Proper indentation helps comprehension**

Summary

- In OOP, we decompose a program into classes
- A class consists of attributes to store data and methods to perform actions
- Once a class has been defined, objects of that class can be created (instantiated)
- Methods are invoked on an object, and may cause the data of the object to change
- A **static** method can be called without creating an object
- A **static** variable is shared by all objects of the class
- Constructor method creates, initializes objects of a class
 - Default constructor has no parameters
- Methods overloading allow us to have multiple methods or constructors with the same name.
 - They must differ in argument signatures (number and/or type of parameters)