# Modular Programming
# Java Classes

# *Introduction*

## *Introduction*

All programs we have written up to this point have the code in a single class/file

Larger programs are written in many classes instead of one. We think of these classes as separate modules

Naming conventions are the same: The class has to have the same name as the file it's in

A class can contain constants, variables, methods, and other classes

## *Introduction*

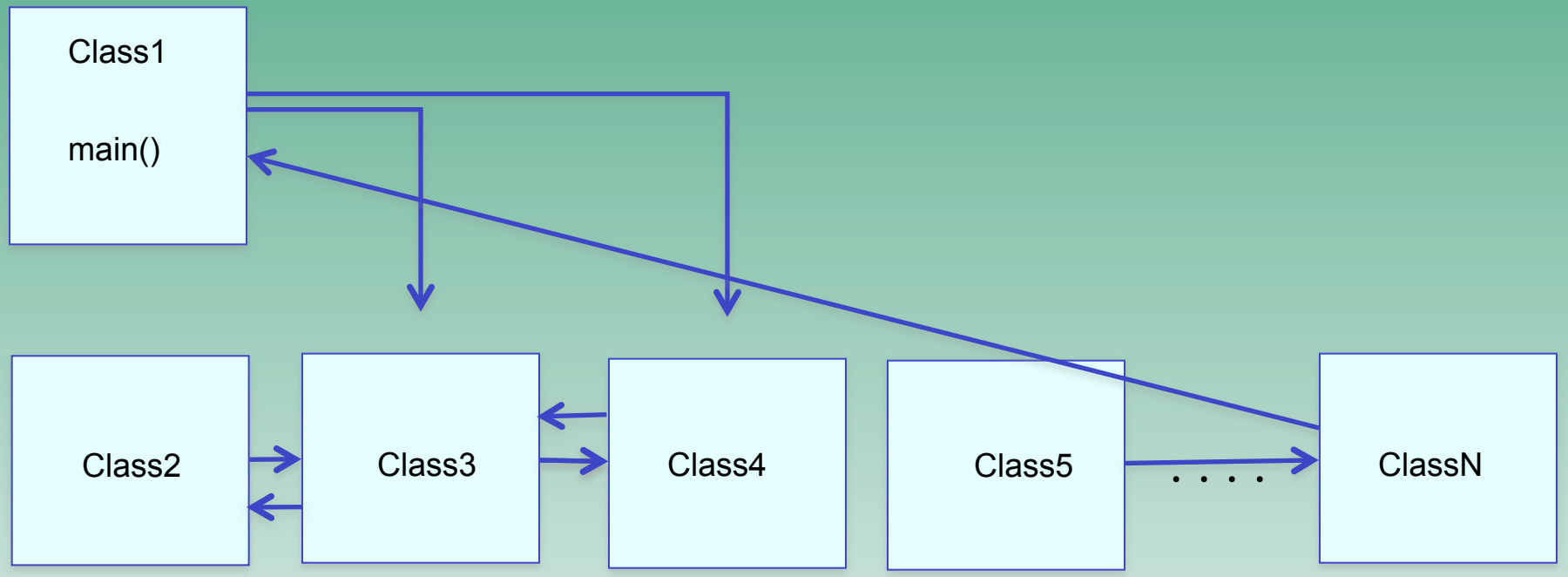
These classes are normally compiled separately

Different people can develop classes independently

When the main method runs, the separately compiled classes can run together. We will keep all compiled classes in the same folder for now

Java uses the classes's filenames to find methods, objects, and variables in other classes. This is the reason why a class' name and its filename have to be the same

# *Introduction*




Any class can reference any other class

The class with the `main()` method starts up the entire running process

# *Introduction*

We look at three different types of Java modules in this course:

1. Classes that hold constant definitions

2. Classes that contain static methods

3. Classes that define objects: Variables and non-static methods

# *1. Classes that hold constant definitions*

We use this type of class to maintain a common location of all constants that are used throughout a program

All `public` constants that are placed in this type of class become accessible to all other classes that reside in the same folder (directory)

This type of class is very simple, convenient, and shareable. It is compiled like any other class

```
public class <ClassName> {
    <constant definition>
    <constant definition>
    <constant definition>
       .
       .
       .
    <constant definition>
}
```

```
public class Definitions {
    public static final int WINDOW_HEIGHT = 300;
    public static final int WINDOW_WIDTH  = 250;

    public static final char BLANK     = ' ';
    public static final char LINE_FEED = 10;

    public static final String PROMPT = "Enter choice: ";

    public static final boolean DONE = true;

    public static final double GRAVITY = 9.8;
    public static final double PI = 3.1415;
}
```

To access any of these constants from another class all we need to do is qualify its name with the name of this class. For example, from our main method we could have an expression like

```
weight = mass * Definitions.GRAVITY;
```

In Java, Constants are written with full capitals by convention

```
public class VolumeOfASphere {
    public static void main(String[] args) {
        double radius = 3.0;

        System.out.print("Volume = ");
        System.out.print(4/3 * Definitions.PI * Math.pow(radius, 2));
    }
}
```

This class uses a constant that resides in `Definitions.class`

The constant `PI` is defined in the file `Definitions.java`

It becomes available to any other class once `Definitions.java` has been compiled to `Definitions.class`

If `Definitions.class` is not found in the current folder, the Java compiler will automatically compile `Definitions.java` into `Definitions.class`

# *Exercise 1*

Create a constants class with the name `Defns` that contains the following constants:

| Name | Type | Value |
|---|---|---|
| FILENAME | String | "messages.txt" |
| DEBUG_ON | boolean | true |
| CR | char | 13 |

Save and compile `Defns.java`

Confirm that your folder contains the files `Defns.java` and `Defns.class`

# *2. Classes that contain static methods*

With this type of class we keep methods of a common type together

This type of class ***does not*** contain a `main()` method

All `public` methods that are placed in this type of class become accessible to all other classes that reside in the same folder (directory)

This type of class can contain `private` methods which are available only to the methods within the class

Simple, convenient, shareable, and compiled like any other class

```
public class <ClassName> {
    <global static variables (optional)>

    <static public/private method>
    <static public/private method>
    <static public/private method>
       .
       .
       .
    <static public/private method>
}
```

```
public class Numerics {
    public static double larger(double m, double n) {
        if (m > n)
            return m;
        else
            return n;
    }
}
```

To access the methods of this class from another class all we need to do is qualify the method name with the name of the class. For example, from our `main()` method we could have an expression like

```
System.out.println(Numerics.larger(28.1, 12));
```

```
public class LargerVolume {
    public static void main(String[] args) {
        double radius = 3.0;
        double height = 7.0;
        double coneVolume   = 1/3 * Definitions.PI *
                              Math.pow(radius, 2) * height;

        double sphereVolume = 4/3 * Definitions.PI *
                              Math.pow(radius, 2);

        System.out.println(Numerics.larger(coneVolume, sphereVolume));
    }
}
```

This program makes use of two static methods: `larger()` and `pow()`

We defined `larger();` `pow()` was defined by someone else

The calls are made the same way: `<ClassName>.<methodName>`

## *Exercise 2*

Turn your `SearchAndSort` into a class of static methods by removing the `main()` method from your `SearchAndSort` class that contains all the methods such as `linearSearch()`, `binarySearch()`, `bubbleSort()`, etc. from our work on searching and sorting.

Save and compile `SearchAndSort.java`

Confirm that your folder contains the files `SearchAndSort.java` and `SearchAndSort.class`

Write a separate class with a `main()` method with a call to `bubbleSort()` and a call to `binarySearch()` to confirm that the systems works correctly

# *3. Classes that define objects*

With this type of class we can *define* the structure of an object

The object is not created yet, but only defined

The object is created when it is *instantiated* in a statement like

```
String s = new String();
```

The class definition contains variables and methods which determine the properties and behaviours of the object

For example, in the statement `s.length()`,

> `s` is the object which is of type `String`
>
> `length()` is a non-static method defined inside the class `String`
>
> in order to use `length()` we must create (instantiate) the object `s` first

In some Java compilers, a string is instantiated automatically, but it is instantiated nevertheless. This can be overriden with a statement like `String s = null;`

## The `String` class with the `length()` method signature

```
public class String {
   ...
   ...
   <variables and methods>
   ...
   public int length() {
       <statements>
       return <value>
   }
   <other methods>
}
```

We can define our own classes

A class allows us to handle many variables with a single object

We place variables and methods inside the class definition

We need to instantiate an object before we can use it

The following slides show the steps necessary to define a class called `Customer`. After it has been defined, an object is instantiated

# *Class name and variables with initialization*

```
public class Customer {
    private String cName = "";
    private String id = "";
    private double balance = 0.00;
    private boolean residesInGTA = false;
}
```

# *Default constructor*

```
public class Customer {
    private String cName = "";
    private String id = "";
    private double balance = 0.00;
    private boolean residesInGTA = false;

    public Customer() {
       cName = "";
       id = "";
       balance = 0.00;
       residesInGTA = false;
    }
}
```

# Constructor with parameters

```
public class Customer {
    private String cName = "";
    private String id = "";
    private double balance = 0.00;
    private boolean residesInGTA = false;

    public Customer() {
       cName = "";
       id = "";
       balance = 0.00;
       residesInGTA = false;
    }

    public Customer(String n, String i, double b, boolean r) {
       cName = n;
       id = i;
       balance = b;
       residesInGTA = r;
    }
}
```

# *The* `toString()` *method*

```
public class Customer {
    <variables>

    <constructors>

    public String toString() {
        return "Name    : " + cName + "\n" +
               "Id      : " + id + "\n" +
               "Balance : " + balance + "\n" +
               "In GTA  : " + residesInGTA;
    }
}
```

# *Instantiation and output*

```
public class CarRepair {
    public static void main(String[] args) {
        Customer customer = new Customer("Roy",
                                         "32413",
                                          4500.75,
                                          true);
        System.out.println(customer);
    }
}
```

## *Exercise 3*

Create a class named `Fruit`. Then define,

A private `String` variable to contain the name of the fruit
A private `String` variable to contain the fruit's country of origin
A private `double` primitive to contain the fruit's price
A default constructor that initializes the variables
A constructor with three parameters for each of the variables
The `toString()` method

Test your class:

Create a class called `FruitMain` that contains the `main()` method
Declare and instantiate an object called `fruit` with the parameters `"orange", "Spain", 0.45`
Use `System.out.print()` to print the contents of `fruit`

## *Exercise 4*

Create a class named `Book`. Then define ,

A private `String` variable to contain the name of the book
A private `String` variable to contain the ISBN of the book
A private `String` variable to contain the author
A default constructor
A constructor with three parameters for each of the class's variables
The `toString()` method

Test your class:

Create a class called `BookMain` that contains the `main()` method
Declare and instantiate an object called `book` and set its name, ISBN and author. Choose a book with a real ISBN
Use `System.out.print()` to print the contents of book

## *Access methods*

Variables inside a class are declared `private`

The variables of an object are not directly accessible from outside the object, but can be accessed indirectly through access methods

For each private variable of a class we need to create 2 public access methods: One to set the variable's value and one to obtain its value

For example, if we want to set the Customer's `balance` to $780.25 from our Customer's class, we need to write a method that takes care of doing this. It is the equivalent to an assignment operation.

```
public void setBalance(double b) {
    balance = b;
}
```

## *Access methods*

If we need to access the `balance` from the Customer object, then we create a method that gives us the balance:

```
public double getBalance() {
    return balance;
}
```

## *Access methods*

In the calling statement we would have the following:

```
Customer c = new Customer();

c.setBalance(780.25);
System.out.println(c.getBalance());
```

## *Exercise 5*

In your `Fruit` class from exercise 3, add `get` and `set` methods for each of the private variables of the class. Then, include statements in a `main()` method that call these new methods.

## *Exercise 6*

In your `Book` class from exercise 4, add `get` and `set` methods for each of the private variables of the class. Then, include statements in a `main()` method that call these new methods.

## *Exercise 7a*

In this exercise we will create a class that performs fraction operations.

Create a class with the header

```
public class Fraction
```

Include three private variables:

1. An integer `numerator` initialized to zero
2. An integer `denominator` initialized to zero
3. A boolean `undefined` initialized to `true`

## *Exercise 7b*

Include a default constructor without any parameters. The constructor initializes the private variables to the same values as at declaration time

Include a constructor with two integer parameters: `n` and `d`

Inside the constructor, assign the `n` parameter to `numerator` and the `d` parameter to the `denominator` variables respectively.

Then, assign to the `undefined` variable the expression `(denominator == 0)`. This will set `undefined` to `true` if the denominator is zero, and to `false` if the denominator is not zero.

Compile the class `Fraction`

## *Exercise 7c*

Include six access methods to set and get the values of `numerator`, `denominator`, and `undefined` variables

Include a `toString()` method that will print a fraction in readable format. For example, when placing a `System.out.println()` call with a numerator of 7 and a denominator of 5, the output should be,

```
7/5
```

If the fraction is undefined, then the output should be the word

```
undefined
```

## *Exercise 7d*

Create a main method, and instantiate four fractions with:

```
a) numerator = 7, denominator = 5
b) numerator = 3, denominator = 2
c) numerator = 0, denominator = 6
d) numerator = 4, denominator = 0
```

Print each fraction using `System.out.println()` and passing the entire fraction object. The output should be:

```
7/5
3/2
0/6
undefined
```

## *Exercise 7e*

This is the code for the multiplication of two fractions. You will need it for the rest of the exercise

```
public Fraction multiply(Fraction other) {
    Fraction result = new Fraction(0, 0);

    if (!undefined && !other.getUndefined()) {
        result.setNumerator(numerator *
                            other.getNumerator());

        result.setDenominator(denominator *
                              other.getDenominator());
    }
    return result;
}
```

## *Exercise 7f*

Implement a method that returns the reciprocal of a fraction if and only if the fraction is defined and its numerator is not zero. Otherwise the method returns an undefined fraction with zeroes in the numerator and the denominator.

### Method signature

```
public Fraction reciprocal()
```

### Algorithm

```
declare and instantiate result as a Fraction: zero over zero
if the fraction is defined and its numerator is not zero {
    set the numerator of result to the denominator of fraction
    set the denominator of result to the numerator of fraction
}
return result
```

## *Exercise 7g*

Implement division of two fractions by using the reciprocal and the multiplication methods

Method signature

```
public Fraction divide(Fraction other)
```

Algorithm

```
return (fraction multiplied by the reciprocal of other)
```

# *Exercise 7h*

Implement addition of two fractions

## Method signature

```
public Fraction add(Fraction other)
```

## Algorithm

(assume that fractions are: `a/b` and `c/d` where `a/b` is the calling fraction and `c/d` is `other`)

```
declare and instantiate result as an undefined Fraction
if both fractions are defined {
    set the numerator of result to a * d + c * b
    set the denominator of result to b * d
}
return result
```

# *Exercise 7i*

Implement subtraction of two fractions

## Method signature

```
public Fraction subtract(Fraction other)
```

## Algorithm

(assume that fractions are: `a/b` and `c/d` where `a/b` is the calling fraction and `c/d` is `other`)

```
declare and instantiate result as an undefined Fraction
if both fractions are defined {
    set the numerator of result to a * d - c * b
    set the denominator of result to b * d
}
return result
```

# *Exercise 7j*

Read Euclid's GCD algorithm: http://en.wikipedia.org/wiki/Euclidean_algorithm

```
// Euclid's Greatest Common Divisor algorithm

private int gcd(int m, int n) {
    while (m != 0 && n != 0) {
        if (m > n)
            m = m - n;
        else
            n = n - m;
    }

    if (m != 0)
        return m;
    else
        return n;
}
```

# *Exercise 7j*

Write the method to simplify a fraction

```
public Fraction simplify()
```

algorithm

```
declare and instantiate result as an undefined Fraction
declare sign as an integer and initialize to 1
if fraction is defined {
    if either numerator or denominator (but not both) is negative {
        sign = -1
    }
    make the fraction positive
    greatestCommonDivisor = gcd(numerator, denominator)
    set numerator of result to numerator / greatestCommonDivisor * sign
    set denominator of result to denominator / greatestCommonDivisor
}
return result
```

## *Exercise 7k*

Revise all operations of the class and include a simplification call where appropriate so that the returned result is a simplified fraction.

Modify the `toString()` method so that:

1. The message `undefined` is printed if the fraction is undefined
2. A single integer is printed if the denominator of the fraction is 1
3. The fraction is printed otherwise

# *Selected Solutions*

# *Solution to Exercise 7a*

```
public class Fraction {
    private int numerator = 0;
    private int denominator = 0;
    private boolean undefined = true;
}
```

# Solution to Exercise 7b

```
public class Fraction {
    private int numerator = 0;
    private int denominator = 0;
    private boolean undefined = true;

    public Fraction() {
        numerator = 0;
        denominator = 0;
        undefined = true;
    }

    public Fraction(int n, int d) {
        numerator = n;
        denominator = d;
        undefined = (denominator == 0);
    }
}
```

# Solution to Exercise 7c

```
public int getNumerator() {
    return numerator;
}

public void setNumerator(int n) {
    numerator = n;
}

public int getDenominator() {
    return denominator;
}

public void setDenominator(int d) {
    denominator = d;
}

public boolean getUndefined() {
    return undefined;
}

public void setUndefined(boolean u) {
    undefined = u;
}
```

# Solution to Exercise 7e

```
public Fraction multiply(Fraction other) {
    Fraction result = new Fraction(0, 0);

    if (!undefined && !other.getUndefined()) {
        result.setNumerator(numerator * other.getNumerator());
        result.setDenominator(denominator * other.getDenominator());
    }
    return result;
}
```

# Solution to Exercise 7f

```
public Fraction reciprocal() {
    Fraction result = new Fraction(0, 0);

    if (!undefined && numerator != 0) {
        result.setNumerator(denominator);
        result.setDenominator(numerator);
    }
    return result;
}
```

# *Solution to Exercise 7g*

```
public Fraction divide(Fraction theOther) {
    return multiply(theOther.reciprocal());
}
```

This example extends the class `Customer` to the sub-class `SearsCustomer`:

```
public class SearsCustomer extends Customer {
    private boolean searsCreditCard = false;

    public SearsCustomer() {
    }

    public SearsCustomer(String c, String i, double b,
                         boolean r, boolean s){
        super(c, i, b, r);
        searsCreditCard = s;
    }

    public void setSearsCreditCard(boolean s) {
        searsCreditCard = s;
    }

    public boolean getSearsCreditCard() {
        return searsCreditCard;
    }

    public String toString() {
        return super.toString() +  "\nCredit Card: " + searsCreditCard;
    }
}
```

The relation between child and parent classes

- The child class has access to the constructors of the parent class: use `super`
- The child class has access to public methods of the parent class
- Only the child has access to protected methods and variables of the parent class
- The child does not have access to private methods and variables of the parent class

# *Exercise 7l (Optional material)*

Define a sub-class of `Fraction` called `MixedNumber`. Using the methods of `Fraction`, define the addition of two positive mixed numbers.

Write a program that adds two mixed numbers. The output should be a mixed number. If the fractional part is zero, only the integer part need be output. (See algorithm on next slide) Examples:

$$1\frac{1}{2}+1\frac{1}{3}=2\frac{5}{6}$$

$$2\frac{3}{7}+5\frac{4}{7}=8$$

$$7\frac{3}{4}+3\frac{2}{7}=7\frac{1}{28}$$

# *Exercise 7l (continued)*

## Algorithm

```
result = declare and instantiate an undefined mixed number
change the calling number to an improper fraction
change the other number to an improper fraction
if both fractions are defined
    Fraction improperResult = add both fractions
    if improperResult is defined then
        result = convert back to a mixed number
    }
}
return result
```