UBC OKANAGAN

COSC 111
Computer Programming I

Final Exam Review

Dr. Firas Moosvi

Acknowledgement: Slides (provided by Dr. Abdallah Mohammed) mainly rely on the materials prepared by Y. D. Liang for the textbook "Introduction to Java Programming, 10th Ed., Pearson Edu. Inc.".

Final Exam Details

The final exam will be held on **Apr 28, 2021, 7:00 PM**

- Check SSC to confirm official date and time!

Weight: the exam worth 20% of the total grade

Time limit: **2.5 hours**

The exam covers **all course material** as indicated in the syllabus and during the lectures.

All students must satisfy ALL conditions to pass the course:

- Pass the Lab component with a grade of at least 50%,
- Pass the Test and Exam components (together) with a grade of at least 50%,
- Pass the Final Exam with a grade of at least 40%.

DISTINGUISHED SPEAKER SERIES

From Big Data to Your Data: How Data Driven Technologies are shaping the future with Nora Young.

Tuesday April 13, 2021

Register here

Final Exam Details

There will be some multiple choice questions, but the majority of the marks will be for coding tasks

The final exam will be:

- Cumulative
- Live (2.5 hours), invigilated, but no proctoring.
- Open book, open-notes, open-web but no cheating sites like Chegg/Course-Hero/Bartleby etc
- IDEs are recommended, they will help you!
- On Canvas, using Gradescope and GitHub

You will NOT be formally tested on Git and Command Line, but you will need those concepts to do the final exam (i.e. accept a GitHub Classroom repository).

The following few slides will go over the contents of the final exam.

What you learned in COSC 111

Terminal

- macOS and Linux: Terminal
- Windows: GitBash

Basic Version Control with git

- `git add -A (OR git add .)`
- `git commit`
- `git fetch`
- `git push`
- `git pull`

Advanced Version Control with git (will NOT be tested)

- Branches
- Merge
- Rebase
- Pull requests, etc…

What you learned in COSC 111

Variables, constants, and data types.

- Casting
- Variables scope
- Primitive vs. reference types

Displaying output using

- `println()`, `print()`, `printf()`

Reading input

- using `Scanner` methods

Operators

- Assignment (=)
- Mathematical (*unary*, e.g., ++, and *binary*, e.g., +)
- Logical (!, &&, ||, ^)
- Relational (e.g., >, <, ==, …)

What you learned in COSC 111

Built-in Methods and Classes

- `Math` class (e.g., `sqrt()`, `pow()`, `sin()`, `random()`, etc)
- `Character` class (e.g., `isDigit()`, `isUpperCase()`, etc)
- `String` class (e.g., `length()`, `charAt()`, `indexOf()`, etc)

  Must know how each method work and what it returns.

Conditional Expressions

- e.g., (x > 5 && y < 3)
  - true or false

Selection

- `if`, `switch`, `?`
  - Must know how to rewrite a statement using another Java keyword.
- `break`

What you learned in COSC 111

Loops

- `for`, `for`-each, `while`, `do-while`
  - Must know how to rewrite a statement using another Java keyword.
  - Must know when a loop will terminate and when it will be infinite.
- `break` and `continue`
- Nested loops

User-defined Methods

- Method calls
- local variables (again)
- Overloading

Arrays (1D and 2D)

- Using loops with arrays
- Passing/returning arrays to/from methods
- Predefined class: `Arrays`
- `System.arraycopy()`

What you learned in COSC 111

Introduction to OOP

- Creating classes and objects (attributes, constructors, methods)
  - Method overloading
  - Constructors vs methods
- Instance vs. `static` class members
- Visibility modifiers (`public`, `private`)
- Objects in the memory as reference variables
- `this` keyword

The three pillars of OOP:

- 1) Encapsulation
  - Getters, setters
- 2) Inheritance (in COSC 121)
- 3) Polymorphism (in COSC 121)

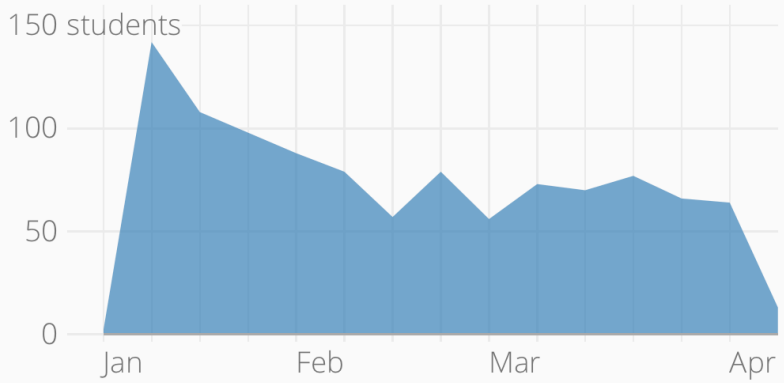
Analytics on Ed Discussion

356	47	4	349	691
Questions	Posts	Announcements	Answers	Comments

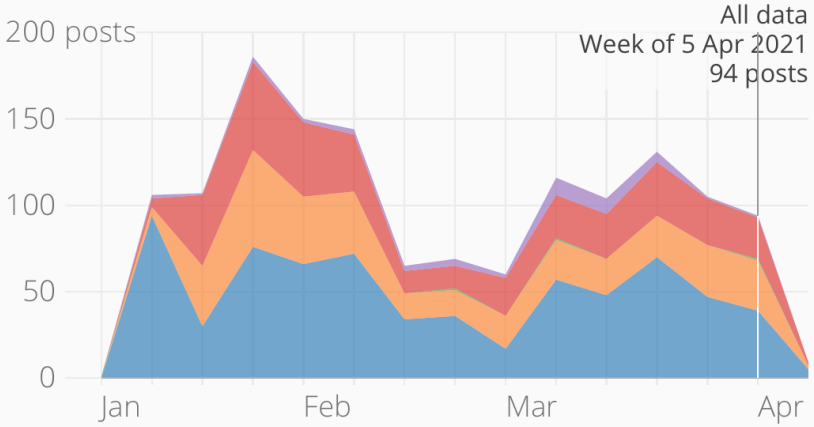
Student participation

Staff participation

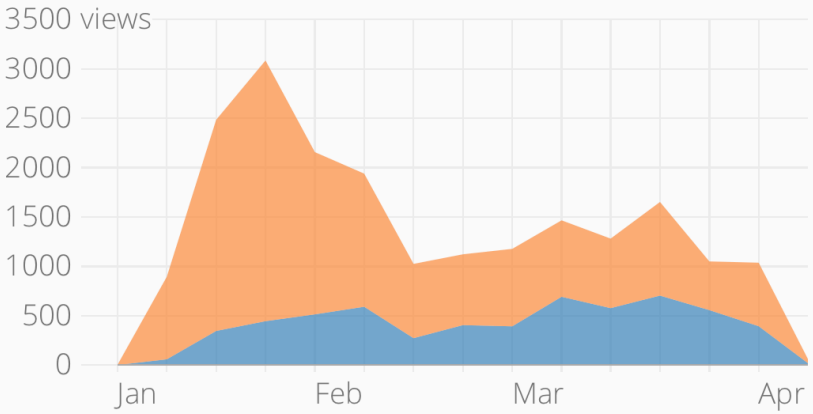
Participation



Discussion activity



Views



Survey on Ed Discussion

Please take 1 minute to complete the survey on Ed Discussion !

https://ubc.ca1.qualtrics.com/jfe/form/SV_40ZjNqgjjZs2QbY

Review Requests

Anonymous Chicken 3 days ago Unresolved

2d arrays with nested for loops

1 Reply Edit Delete •••

Anonymous Chicken 3 days ago Unresolved

more info on how for loops and while loops work

1 Reply Edit Delete •••

Anonymous Magpie 3 days ago Unresolved

Have we covered inheritance? I dont remember anything about that.

1 Reply Edit Delete •••

A Alison Simpson 3 days ago Unresolved

while loops, and do while loops confuse me.

Also using getters and setters for object orientated programming

1 Reply Edit Delete •••

Anonymous Quail 2 days ago Unresolved

More info on constructor and reference objects, please. :)

2 Reply Edit Delete •••

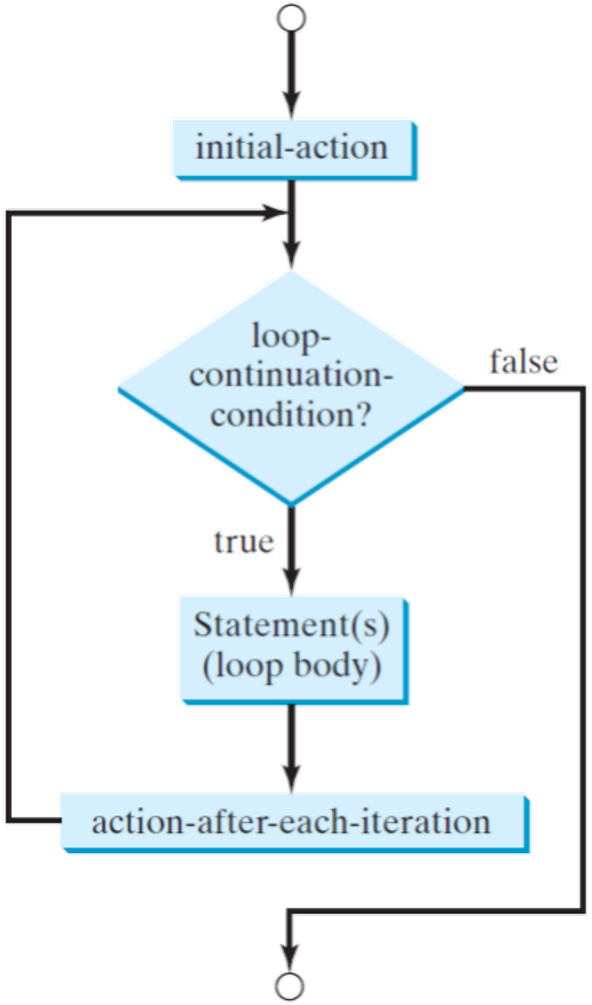
Anonymous Zebra a day ago Unresolved

test 4 q8?

♡ Reply Edit Delete •••

For Loops and While Loops

for Loops



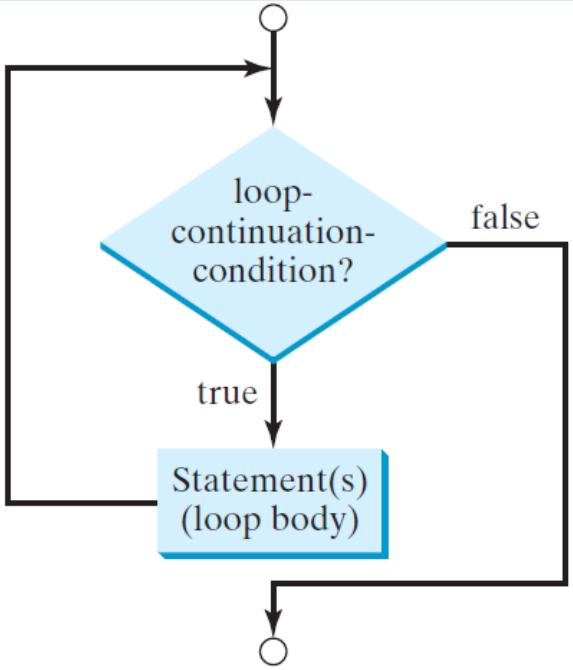
count is incremented (to keep record of how many times we went through the loop

count acts as a counter

Condition is checked BEFORE running the loop

```
for(int counter=0; counter<100; counter++){

    System.out.println("Welcome to Java");

}
```

while loop



count acts as a counter

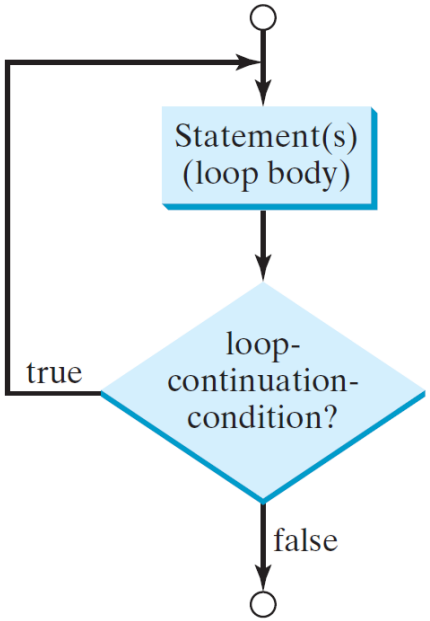
Condition is checked before running the loop

count is incremented (to keep record of how many times we went through the loop

```
int count = 0;

while (count < 100) {

    System.out.println("Welcome to Java");

    count++;

}
```

do-while Loop

A **`do-while`** loop is the same as a **`while`** loop except that it executes the loop body first and then checks the loop continuation condition.



count acts as a counter

count is incremented (to keep record of how many times we went through the loop

Condition is checked AFTER running the loop

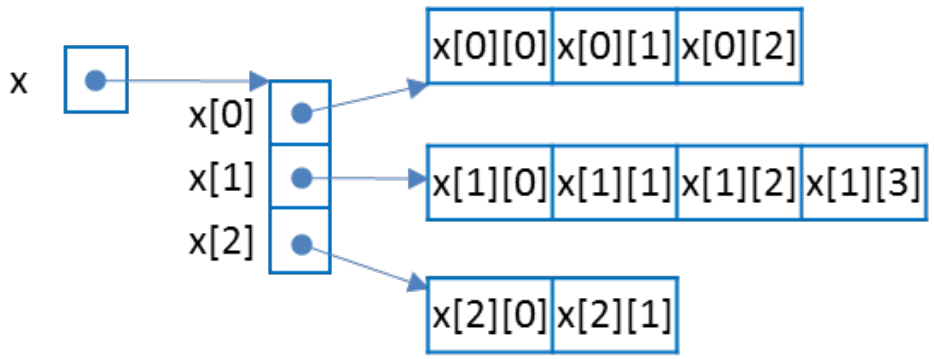
```
int count = 0;
do {
      System.out.println("Welcome!");

      count++;
} while (count < 100);
```

2D Arrays with Nested for loops

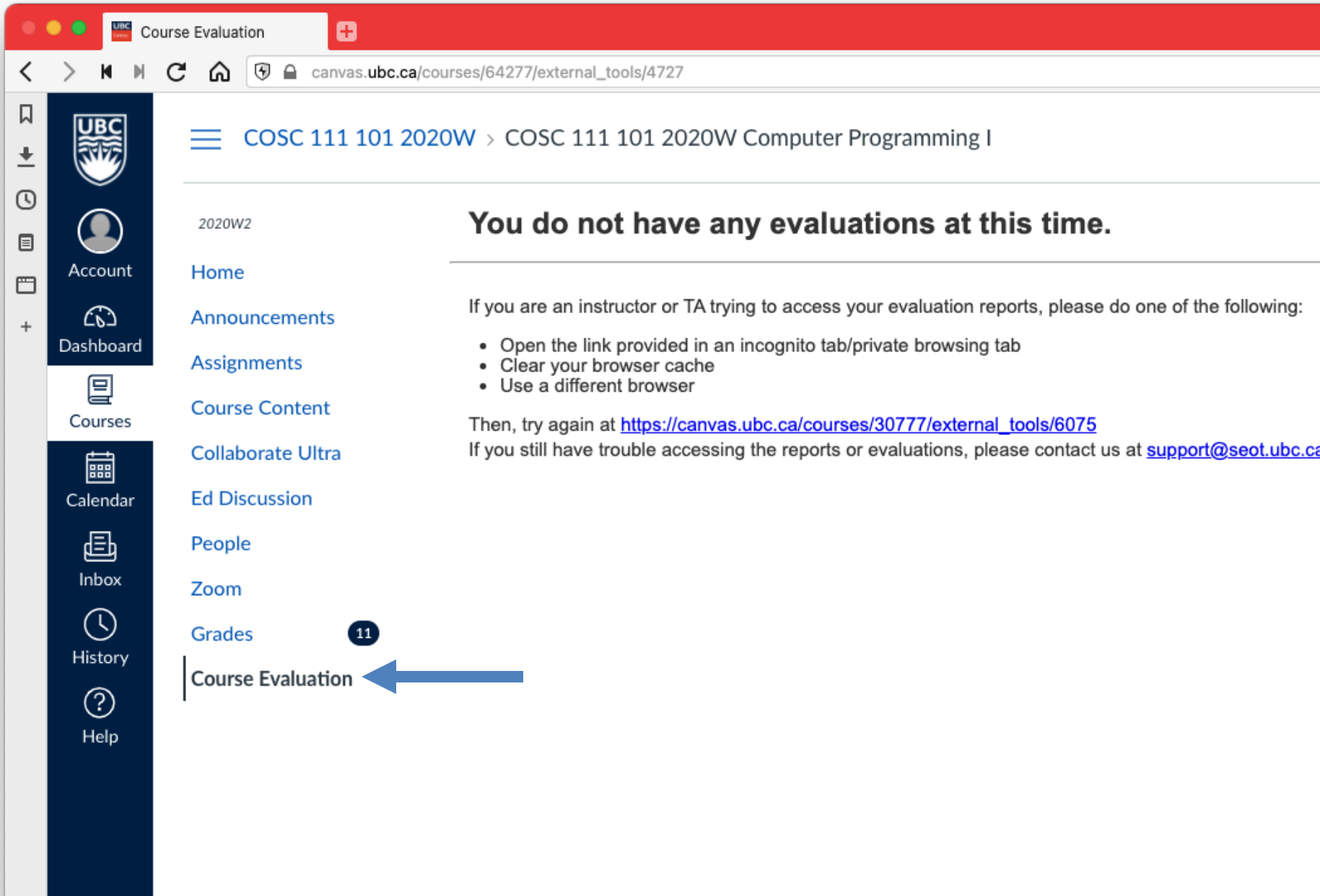
Processing 2D Array

Similarly to 1D arrays, you may use for loops for accessing 2D arrays. A common syntax to process **all elements evenly** is as follows:

```
for (int r = 0; r < x.length; r++){
    for (int c = 0; c < x[r].length; c++){
        //statements that are applied to all elements evenly
    }
}
```



Student Evaluations of Teaching



Student Evaluations of Teaching

Response Rate

	Responded	Invited	% Rate
Students	32	149	21.48%

Let's take 5 minutes to complete the course evals now

Test 4 Q8

Write a method that takes a string made of uppercase letters and spaces and outputs a scrambled version of that string.

To scramble the string, take each letter of the string and replace it with a letter 2n letters away. ie, if n=5, replace the letter A with the letter K. Letters at the end of the alphabet will wrap around, ie, the letter Y becomes I.

Use the following as your method signature:

```
public static string encrypt(string message, int n)
```

Also write the decrypt method as well. This means that sending the output of the encrypt function as the input of the decrypt method, will result in the original message. Use the following method signature:

```
public static string decrypt(string message, int n)
```

Test 4 Q8

1 Answer

R

Ryan DeWolfe TA

4 days ago

Hi,

Like most programming tasks, there are a few different ways to accomplish this. I will outline one here.

- Take the int value of the letter and get it between 0 and 25 (with 0==a and 25==z)
- Add the encryption value.
- Take the remainder using % to keep the value between 0 and 25
- Readjust this value back to the proper ascii character value and cast to a Char.

Source: Ryan DeWolfe (COSC 111 TA)

Constructors in OOP

Constructors

What if I want to initialize objects as I create them?

- **Example:**

```
Farmer f1 = new Farmer();

f1.name = "Mark";
f1.weight = 60.5;
f1.x = 20;
f1.y = 10;
```

```
Farmer f1 = new Farmer("Mark", 60.5, 20, 10);
```

Constructors, cont'd

Constructors play the role of **initializing objects**.

Constructors are a **special kind of method.**

They have 3 peculiarities:

- Constructors must have the **same name as the class itself**.
- Constructors **do not have a return type** -- not even void.
- Constructors are invoked using the **`new`** operator **when an object is created**.

The Default Constructor

A **default constructor** is provided automatically only if no constructors are explicitly defined in the class.

It sets the attributes to their default values:

- String → null
- Numeric → zero
- Boolean → false

In the previous example, the programmer included a four-argument constructor, and hence the default constructor was not provided.

Two Constructors

```
class Farmer {
      //instance variables
      ...
      //constructors
      Farmer(){    //now we have a zero-arg constructor
      }
      Farmer(String fname, int fweight, int fx, int fy){
         name = fname;
         weight = fweight;
         x = fx;
         y = fy;
      }
      //methods
      ...
      public static void main(String[] args) {
            Farmer f1 = new Farmer("Mark", 60.5, 20, 10);
            Farmer f2 = new Farmer("Jessa", 51.4, 17, 13);
            Farmer f3 = new Farmer();  // No error!!
      }
}
```

Getters and Setters for OOP

Data Field Encapsulation

It is preferred to declare the data fields **private** in order to

- protect data from being mistakenly set to an invalid value
  - e.g., c1.radius = -5 //this is logically wrong
- make code easy to maintain.

You may need to provide two types of methods:

- A **getter method** (also called an **'accessor'** method):
  - Write this method to make a private data field accessible.
- A **setter method** (also called a **'mutator'** method)
  - Write this method to allow changes to a data field.

Usually, constructors and methods are created public unless we want to “hide” them.

Practice

Write a public class **SimpleCircle** which has:

- an instance variable double radius.
- a no-argument constructor that sets radius to 10.
- a one-argument constructor that sets radius to a given value.
- a method setRadius that changes the radius to a given value.
- two methods getArea that returns the area.

Test your class by creating three instances of SimpleCircle and invoke their different methods.

Solution

```
class SimpleCircle {
  //Attributes
  private double radius;
  //Constructors
  public SimpleCircle() {
    setRadius(1);
  }
  public SimpleCircle(double radius){
    setRadius(radius);
  }
  //Methods
  public double getRadius() {
    return radius;
  }
  public void setRadius(double r){
    if (radius >= 0)
        radius = r;
  }
  public double getArea() {
    return radius*radius*Math.PI;
  }
}
```

```
public class TestCircle {
  public static void main(String[] args) {
    Circle c1 = new Circle();
    Circle c2 = new Circle(5.0);

    System.out.println("For a circule of radius "
      + c1.radius+", the area is " + c1.getArea());
    System.out.println("For a circule of radius "
      + c2.radius+", the area is " + c2.getArea());
    c1.setRadius(100);
    System.out.println("For a circule of radius "
      + c1.radius+", the area is " + c1.getArea());
  }
}
```

TestCircle is the main class. Its sole purpose is to test the second class.

We could include the main method in SimpleCircle class, and hence SimpleCircle will be the main class.

Practice Questions

Example 1a

What is the value of these expression?

3 + 4 * 4 > 5 * (4 + 3) - 1 & (4 + 3 > 5) ;

Answer: apply the precedence rules as follows:

- Parentheses
  3 + 4 * 4 > 5 * 7 - 1 & (4 + 3 > 5)
  3 + 4 * 4 > 5 * 7 - 1 & (7 > 5)
  3 + 4 * 4 > 5 * 7 - 1 & true
- Mathematical
  19 > 34 & true
- Relational
  false & true
- Logical
  false

Example 1b

What is the value of these expression?

3 + 4 * 4 > 5 * (4 + 3) - 1 **&&** (4 - 3 > 5) ;

Answer: apply the precedence rules as follows:

Left part first:

- WHY?

- Parentheses 3 + 4 * 4 > 5 * 7 - 1 && (4 - 3 > 5)
- Mathematical 19 > 34 && (4 - 3 > 5)
- Relational false && (4 - 3 > 5)
- Logical false

Example 1c

What is the value of this expression?

```
//assume s = “abc”

s.length() + s.substring(0, 2)
```

Answer :

- s.length() + s.substring(0, 2) = 3 + "ab" = "3ab"

Example 2

For 2D arrays:

- Write methods to find the sum (or max, min, average) of
  - all elements
  - each row (or column)

- Write a method to print the array
  - Using two for loops
  - Using only ONE for loop

Test your code with the array below:

```
int[][] arr = { { 14, 11, 13, 12 },
                { 18, 15, 13, 13 },
                { 19, 16, 15, 17 } };
```

Example 3

String processing: write code to process a given string

- e.g. a method to get the index of the n^{th} occurrence of a given letter

```
int nIndexOf(String s, char ch, int n)
```

- e.g. a method to count the number of occurrences of a given letter

```
int countChar(String s, char ch)
```

- e.g. Write a program that prompts the user to enter a string and displays the characters at odd positions.
  - If user enters *Okanagan*, the output would be *Oaaa*

Example 4a

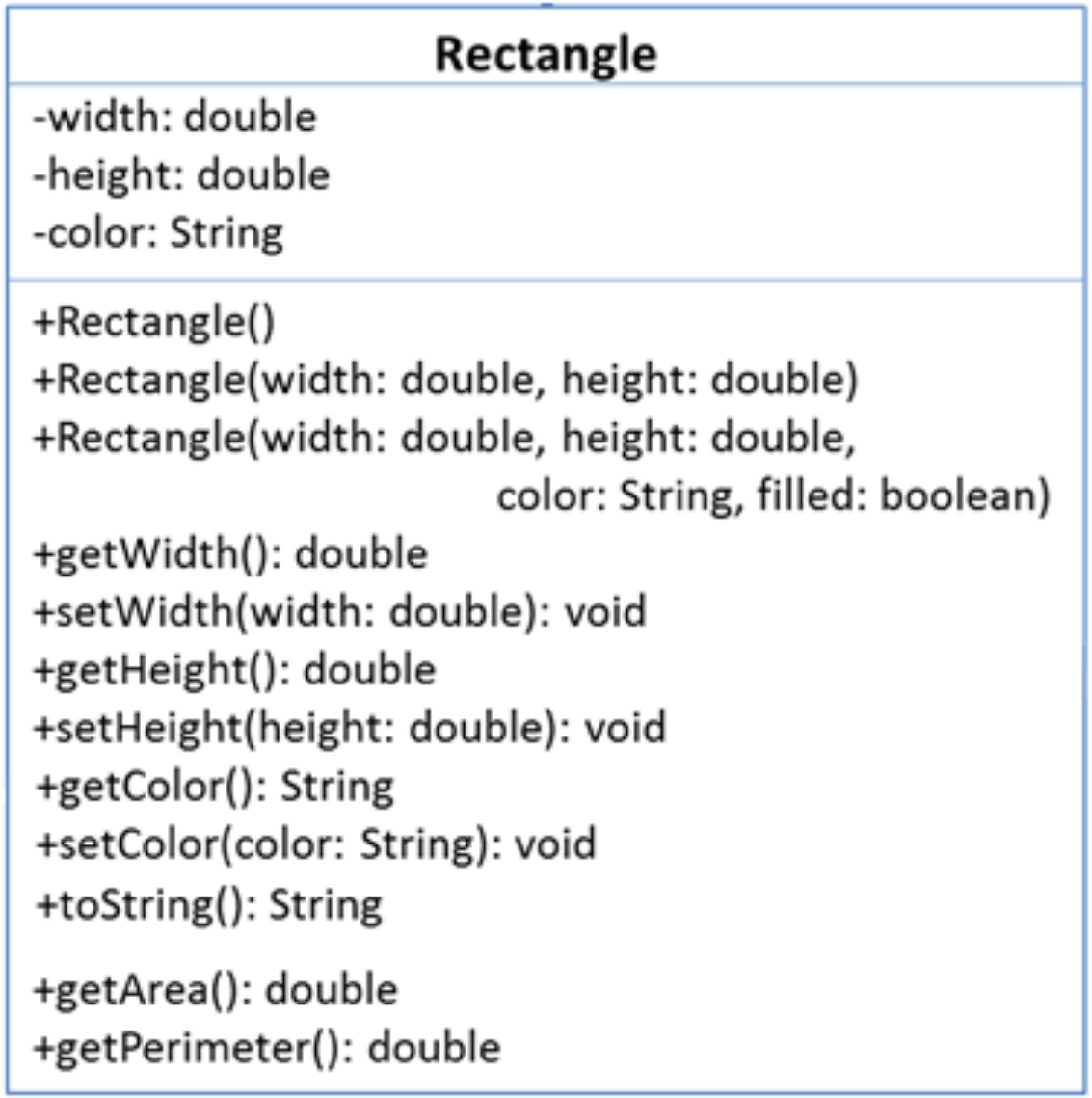
Develop a class X to the following specs: …

Example 1: develop the Rectangle class →

- Restrictions: color can only be “Red”, “Blue”, or “Green’

Example 2: create a robot class. A robot has an id, year, location, memory (as an array of integer).

Example 3: create a spaceship class for a game. Attributes: location, health (and lives?), weapons (array), alive (if not destroyed), etc.



UBC OKANAGAN

COSC 111
Computer Programming I

Preview of Inheritance

(not on COSC 111 Final)

Dr. Firas Moosvi

Acknowledgement: Slides (provided by Dr. Abdallah Mohammed) mainly rely on the materials prepared by Y. D. Liang for the textbook "Introduction to Java Programming, 10th Ed., Pearson Edu. Inc.".

The Three Pillars of OOP

Inheritance (Preview of COSC 121)

Inheritance Overview

Inheritance is a mechanism for enhancing and extending existing, working classes.

- In real life, you inherit some of the properties from your parents when you are born. However, you also have unique properties specific to you.
- In Java, a class that extends another class inherits some of its properties (methods, instance variables) and can also define properties of its own.

`extends` is the key word used to indicate when one class is related to another by inheritance.

Syntax: `class` *`subclass`* **`extends`** *`superclass`*

- The ***superclass*** is the existing, parent class.
- The ***subclass*** is the new class which contains the functionality of the superclass plus new variables and methods.
- A subclass may only inherit from ***one*** superclass.

Why use inheritance?

The biggest reason for using inheritance is to ***re-use code.***

- Once a class has been created to perform a certain function it can be re-used in other programs.
- Further, using inheritance the class can be extended to tackle new, more complex problems without having to re-implement the part of the class that already works.

The alternative is copy and paste which is bad, especially when the code changes.

Example:

- in the `Circle` and `Rectangle` classes we implemented a few slides ago, there was a lot of code redundancy (e.g. `setColor()` was exactly repeated).
- A better solution is to have a superclass, e.g. `Shape`, that has the common code and then have `Circle` and `Rectangle` inherit from `Shape`.

What is inherited?

When a subclass inherits (or extends) a superclass:

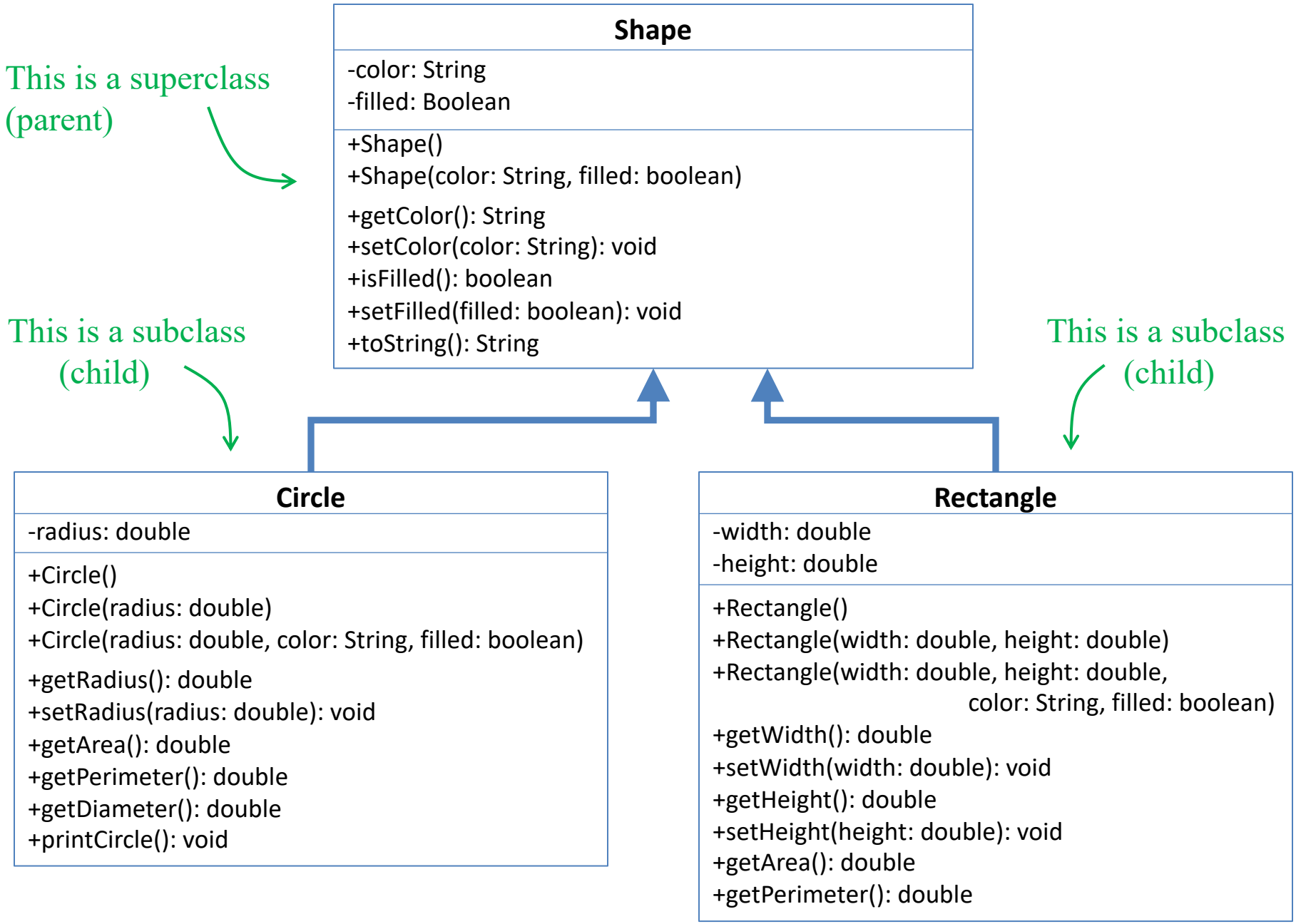
Instance variable inheritance:

- All instance variables of the superclass are inherited by the subclass.
  - However, if a variable is **`private`**, it can only be accessed using methods defined by the superclass.

Method inheritance:

- All superclass methods are inherited by the subclass, but they may be ***overridden***.

Inheritance Example



Inheritance Example, cont.

Shape is the superclass (parent)

```
public class Shape {
    private String color;
    private boolean filled;

    public Shape() {color = "white";}
    public Shape(String color, boolean filled) {
        this.color = color;
        this.filled = filled;
    }
    public String getColor() {return color;}
    public void setColor(String color) {this.color = color;}
    public boolean isFilled() {return filled;}
    public void setFilled(boolean filled) {this.filled = filled;}

    public String toString() {
        return "Color is " + color + ". Filled? " + filled;
    }
}
```

Inheritance Example, cont.

Rectangle is the subclass.
Its parent is Shape

```
public class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle() { this(1.0, 1.0); }
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public Rectangle(double w, double h, String color, boolean filled) {
        setWidth(w);
        setHeight(h);
        setColor(color);
        setFilled(filled);
    }
    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }
    public double getArea() { return width * height; }
    public double getPerimeter() { return 2 * (width + height); }
}
```

Inheritance Example, cont.

This is a test program!

```
public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(1);
        System.out.println("A circle\n" + circle.toString());
        System.out.println("The color is " + circle.getColor());
        System.out.println("The radius is " + circle.getRadius());
        System.out.println("The area is " + circle.getArea());
        System.out.println("The diameter is " + circle.getDiameter());

        Rectangle rectangle = new Rectangle(2, 4);
        System.out.println("\nA rectangle\n" + rectangle.toString());
        System.out.println("The area is " + rectangle.getArea());
        System.out.println("The perimeter is " + rectangle.getPerimeter());
    }
}
```

The output

```
A circle
Color is white. Filled? false
The color is white
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0

A rectangle
Color is white. Filled? false
The area is 8.0
The perimeter is 12.0
```

What can you do in a subclass?

A subclass inherits from a superclass. You can:

- **Use** inherited class members (properties and methods).
- **Add** new class members.
- **Override** instance methods of the superclass
  - to modify the implementation of a method defined in the superclass
  - the method must be defined in the subclass using the same signature and the same return type as in its superclass.
- **Hide** static methods of the superclass
  - By writing a new *static* method in the subclass that has the same signature as the one in the superclass.
- **Invoke** a superclass constructor from within a subclass constructor
  - either ***implicitly***
  - or ***explicitly*** using the keyword super

Overriding methods

Overriding vs. Overloading

Overridden methods are in different classes related by inheritance; **overloaded** methods can be either in the same class or different classes related by inheritance.

Overridden methods have the same signature and return type; **overloaded** methods have the same name but a different parameter list.

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Clicker Question

In which OOP pillar does an object acquire all the attributes and behaviors of the parent object?

A. Encapsulation

B. Inheritance

C. Polymorphism

D. None of the above

Clicker Question

What is subclass in java?

A. a class that extends another class

B. a class declared inside a class

C. a class that uses the keyword `sub` in its header

D. Both above.

E. None of the above.

Clicker Question

What is/are the advantage(s) of inheritance in Java?

A. Code re-usability

B. Save development time

C. Class extendibility

D. All of the above

Clicker Question

Which of the following is not inherited?

A. Instance variables

B. Constructors

C. Method

D. Both (B) and (C)

E. None of the above

Clicker Question

What is the output?

A. 0

B. 1

C. 2

D. 3

E. Error

```
class A {
   public int x;
   public void display() {
      System.out.println(x);
   }
}
class B extends A {
   public int y;
   public void display() {
      System.out.println(y);
   }
}

class Test {
   public static void main(String args[]){
      B b = new B();
      b.x = 1;
      b.y = 2;
      b.display();
   }
}
```

Clicker Question

What is the output?

A. 0

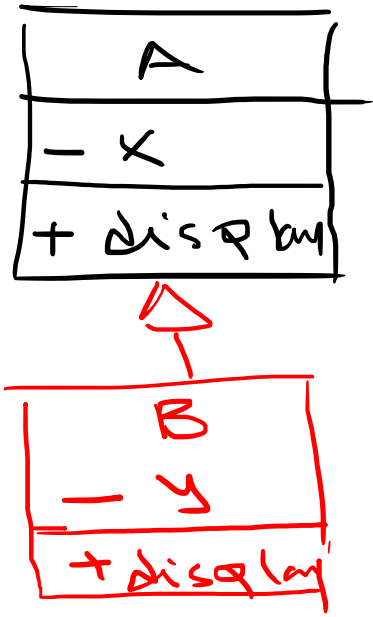
B. 1

C. 2

D. 3

E. Error

```
class A {
   private int x;
   public void display() {
      System.out.println(x);
   }
}
class B extends A {
   private int y;
   public void display() {
      System.out.println(x+y);
   }
}

class Test {
   public static void main(String args[]){
      B b = new B();
      b.display();
   }
}
```



Be careful!

`this` and `super` keywords

The `this` Keyword

The `this` keyword is the name of a reference that an object can use to refer to itself.

Uses:

- To reference class members within the class.
  - Class members can be referenced from anywhere within the class
  - Examples:
    - this.x = 10;
    - this.amethod(3, 5);
- To enable **a constructor to invoke another constructor** of the same class.
  - A constructor can only be invoked from within another constructor
  - Examples:
    - this(10, 5);

The `super` Keyword

The keyword `super` refers to the superclass of the class in which super appears.

Uses:

- To reference class members in the superclass.
  - Example:
    - super.amethod(3, 5);
    - super.toString();
- To enable **a constructor to invoke another constructor** of the superclass.
  - A constructor can only be invoked from within another constructor
  - Examples:
    - super(10, 5);

Superclass Constructors

Explicit & implicit calling of superclass constructor

If no constructor is called within a given constructor, Java implicitly calls the super constructor. For example, the following two segments of code are equivalent:

```
class A{
  public A(){
    System.out.print(1);
  }
}

class B extends A{
  public B(){
    System.out.print(2);
  }
}
```

```
class A{
  public A(){
    System.out.print(1);
  }
}

class B extends A{
  public B(){
    super();
    System.out.print(2);
  }
}
```

Output of
B b = new B();
is 12

- CAUTION: You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as **constructor chaining**.

```
class Person {
  public Person() {System.out.print(1);}
}
class Employee extends Person {
  public Employee() {
    this(2);
    System.out.print(3);
  }
  public Employee(int n) {System.out.print(n);}
}
class Faculty extends Employee {
  public Faculty() {System.out.print(4);}
}
```

```
public static void main(String[] args) {
   Faculty f = new Faculty(); //output is 1234
}
```

Example on the Impact of a Superclass without no-arg Constructor

What is wrong with the code below?

```
public class Fruit {
    String name;
    //Constructors
    public Fruit(String name) {
        this.name = name;
    }
}
```

```
public class Apple extends Fruit{

}
```

Clicker Question

Which of these keywords is used to call a no-arg constructor of a superclass named **`Shape`** from its subclass?

A. `Shape()`

B. `this.Shape()`

C. `super.Shape()`

D. `this()`

E. `super()`

`final` modifier

The `final` Modifier

A **`final`** local variable is a constant inside a method.

The **`final`** class cannot be extended:

```
final class Math {
    ...
}
```

The **`final`** method cannot be overridden by its subclasses.

Clicker Question

Which of the following is FALSE

A. final class cannot be inherited

B. final method can be inherited

C. final method can be overridden

D. final variable cannot be changed.

Clicker Question

Which class cannot be extended?

A. suerclass

B. subclass

C. final class

D. Object class

E. abstract class

Visibility Modifiers Revisited

Visibility Modifiers

Access modifiers are used for controlling levels of access to class members in Java. We shall study two modifiers:

`public`,

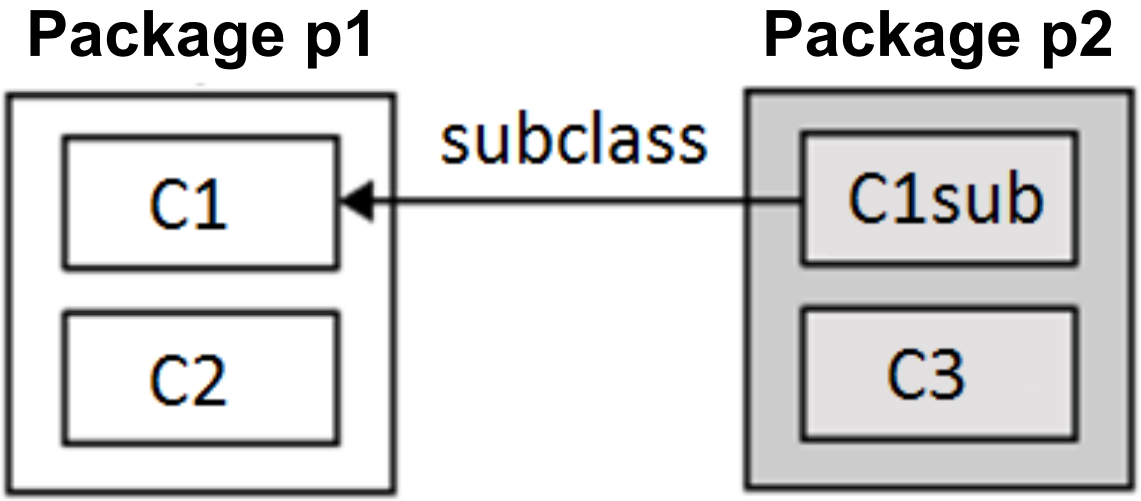
- The class, data, or method is visible to any class in any package.

`Private:`

- The data or methods can be accessed only by the declaring class.

If no access modifier is used, then a class member can be accessed by any class in the same package.

Visibility Modifiers



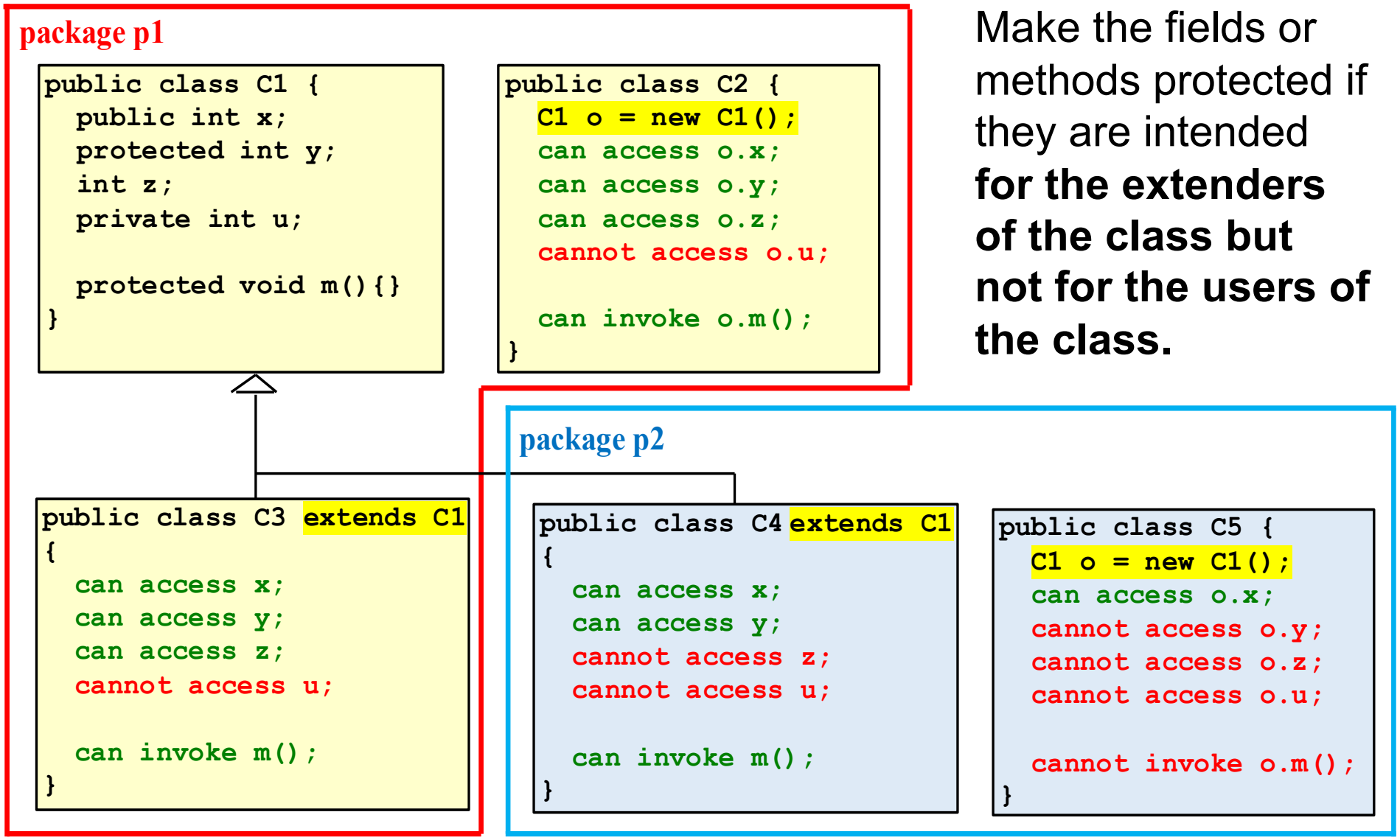
Visibility of a class member in C1

Modifier	C1	C2	C1sub	C3
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
Private	Yes	No	No	No

NOTE

Java 9 introduces a new feature: Java modules, which allows for more accessibility levels (e.g. public to module only instead of to all) but we won't discuss it in this class.

Visibility Modifiers



Make the fields or methods protected if they are intended **for the extenders of the class but not for the users of the class.**

Clicker Question

What is the output?

A. 0

B. 1

C. 2

D. 01

E. Error

```
class A {
   public int x;
   private int y;
}

class B extends A {
   public void display(){
      super.y = super.x + 1;
      System.out.println(super.x+super.y);
   }
}

class Q {
   public static void main(String args[]){
      B b = new B();
      b.display();
   }
}
```

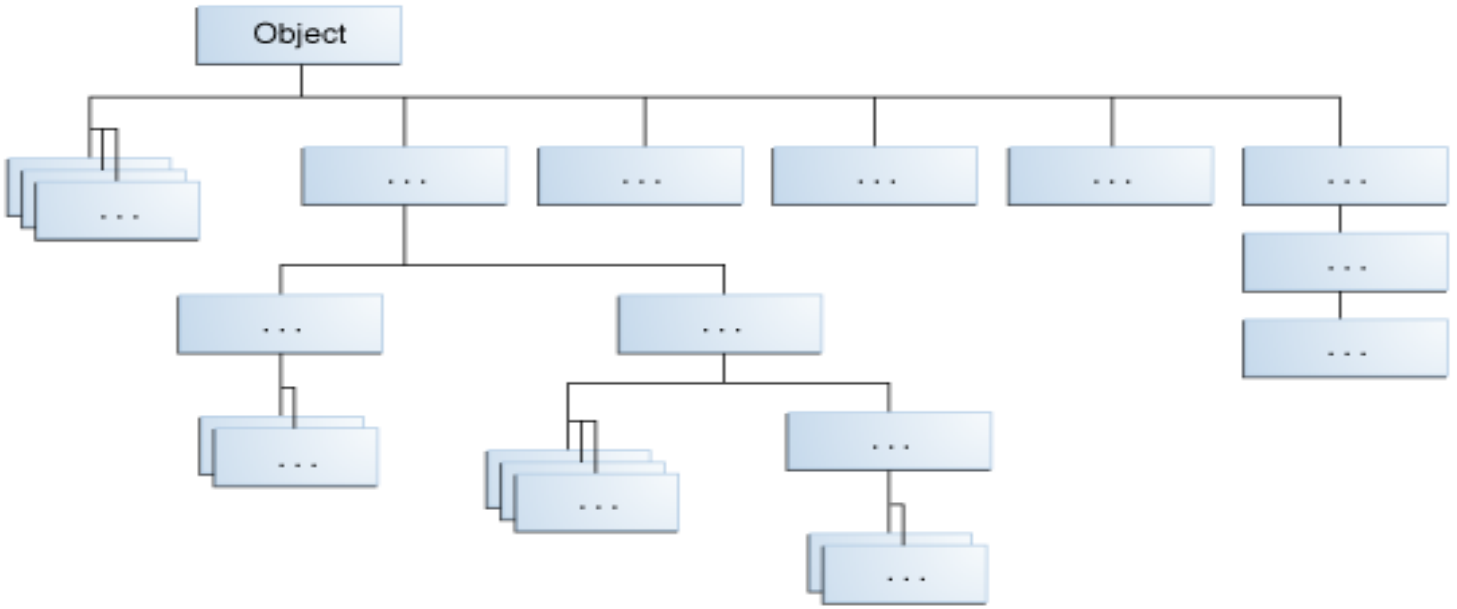
A Subclass Cannot Weaken the Accessibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

The `Object` Class and Its Methods

The `Object` class

Classes in Java are descendants of **`java.lang.Object`** class



Source: oracle.com

Several methods are inherited from **`Object`** such as:

- **`public String toString()`**
  - Returns a string representation of the object.
- **`public boolean equals(Object obj)`**
  - Indicates whether some other object is "equal to" this one
- …

The `toString()` method

The `toString()` method returns a string representation of the object.

Usually you should **override the `toString` method** so that it returns a descriptive string representation of the object.

- For example, the `toString` method in the `Object` class was overridden in the `Shape` class presented earlier as follows:

```
public String toString() {
    return "Color is " + color + ". Filled? " + filled;
}
```