COMP 110-001
Final Exam Review

Yi Hong

June 15, 2015

Final Exam

- Wednesday, June 17th, 8am – 11am
- The final exam will be similar to our midterm, the number of questions will be doubled
- 20% of your grade

Computer Basics

- Hardware and software
- CPU and memory
- Bit and byte
- Program and algorithm
- Compiler and interpreter

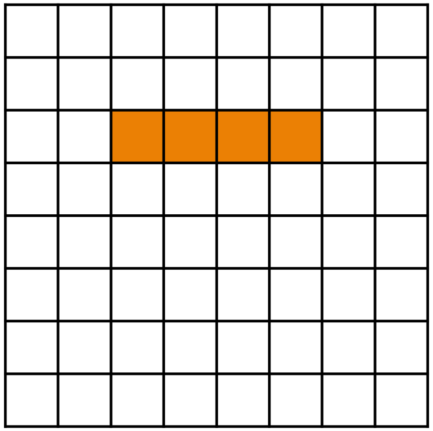
Variables

- A variable is a program component used to store or represent data
  - A variable corresponds to a location in memory
  - Data types: primitive type and class type
- Legal identifier
  - Letters, digits (0-9), and the underscore (_)
  - First character *cannot* be a digit
  - No spaces
  - You cannot name your variables using keywords
  - Java is case sensitive

Variables of a Primitive Type

- A data value is stored in the location assigned to a variable of a primitive type

```
int sum;

sum = 4;

sum = sum + 1;
```

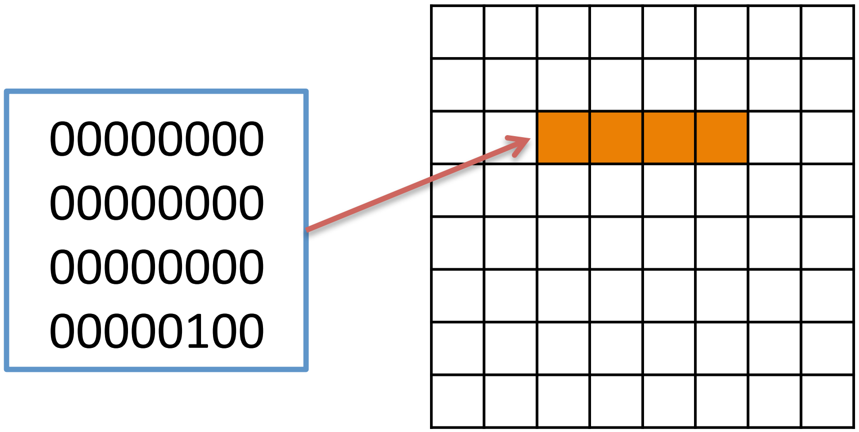


Memory

Variables of a Primitive Type

- A data value is stored in the location assigned to a variable of a primitive type

```
int sum;

sum = 4;

 sum = sum + 1;
```



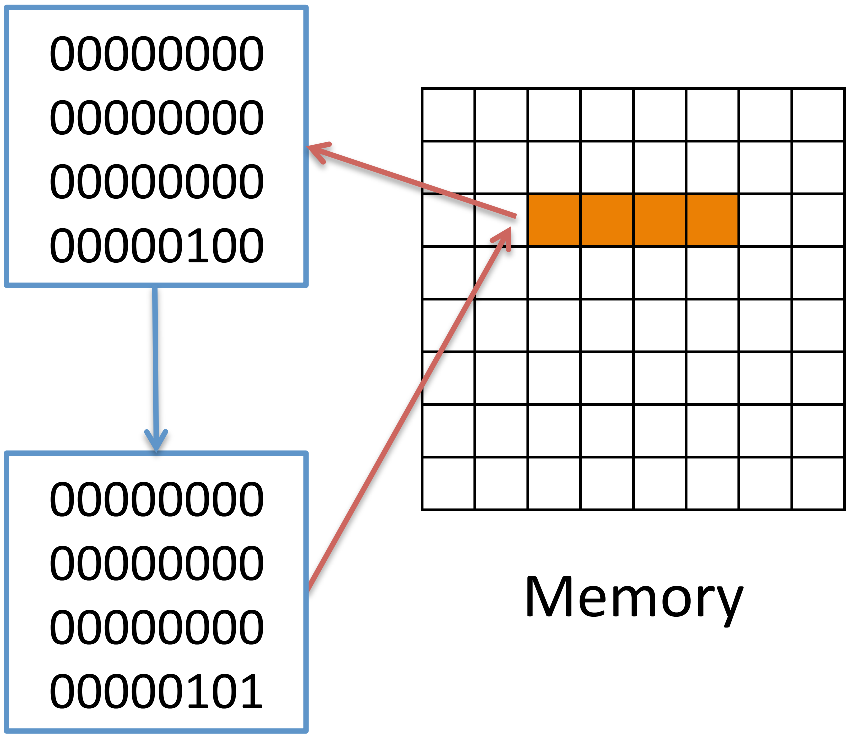
00000000
00000000
00000000
00000100

Memory

Variables of a Primitive Type

- A data value is stored in the location assigned to a variable of a primitive type

```
int sum;

sum = 4;

sum = sum + 1;
```



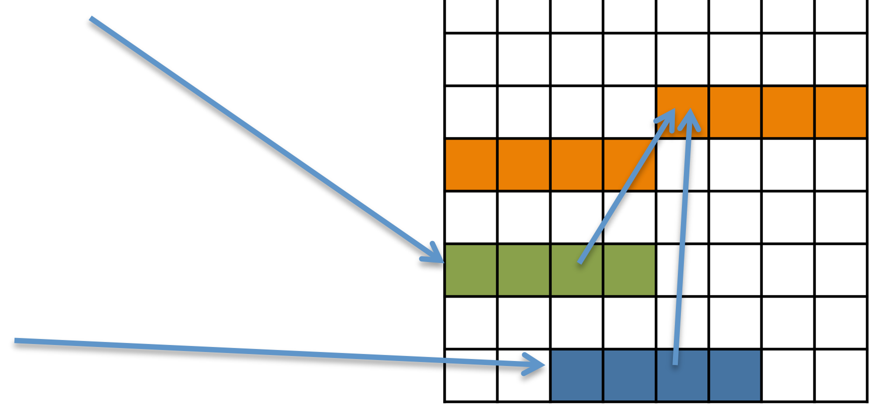
00000000
00000000
00000000
00000100

00000000
00000000
00000000
00000101

Memory

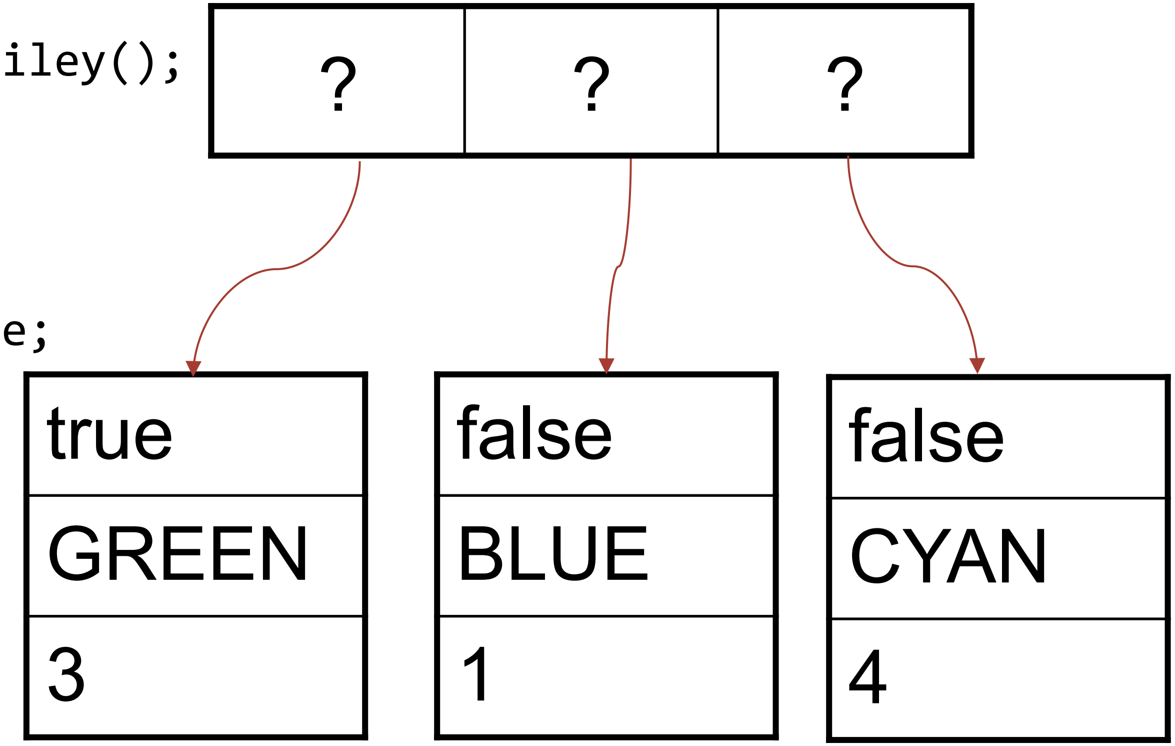
Variable of Class Types

```
Student anna = new Student();
anna.PID = 1234;
anna.year = 3;

Student aCopy = anna;
aCopy.year = 4;

System.out.println( anna.year );
```



Arrays of Objects

```
Smiley[] smilies = new Smiley[3];
for (int i = 0; i < smilies.length; i++)
{
    smilies[i] = new Smiley();
}

smilies[0].bSmile = true;
……
```



Type Casting

- Implicit converting
  - Byte -> short -> int -> long -> float -> double
  - Automatically cast types when they are not match
  - E.g.: double var = 3 / 2;
- Explicit casting
  - Explicitly write the type casting
  - E.g.: int var = (int)(3.0 / 2.0);

String

- A Class Type
- Objects of String class can be defined as:
  - String myString = “UNC is Great!”;
  - Or String myString = new String(“UNC is Great!”);
- Each String object consists of
  - A sequence of characters (char)

String	U	N	C		i	s		G	r	e	a	t	!
Indices:	0	1	2	3	4	5	6	7	8	9	10	11	12

String’s Methods

String myString = “UNC is Great!”

U	N	C		i	s		G	r	e	a	t	!
0	1	2	3	4	5	6	7	8	9	10	11	12

int strLength = myString.**length**();	int, 13
char strFirstLetter = myString.**charAt**(0);	char, ‘U’
boolean bCheck = myString.equalsIgnoreCase(“unc is great!”);	boolean, true
String subStr1 = myString.**substring**(0, 3);	String, “UNC”
String subStr2 = myString.**substring**(7);	String, “Great!”
int pos1 = myString.indexOf(“ ”);	int, 3
int pos2 = myString.lastIndexOf(“ ”);	int, 6

String Concatenation

- String name = “May”;
- String sentence;
- sentence = “My dog’s name is ” + name;

My dog’s name is May

Branch Statements: if-else

- A branching statement that chooses between two possible actions

```
if (Boolean_Expression)
   { statement 1; }
else
   { statement 2; }
```

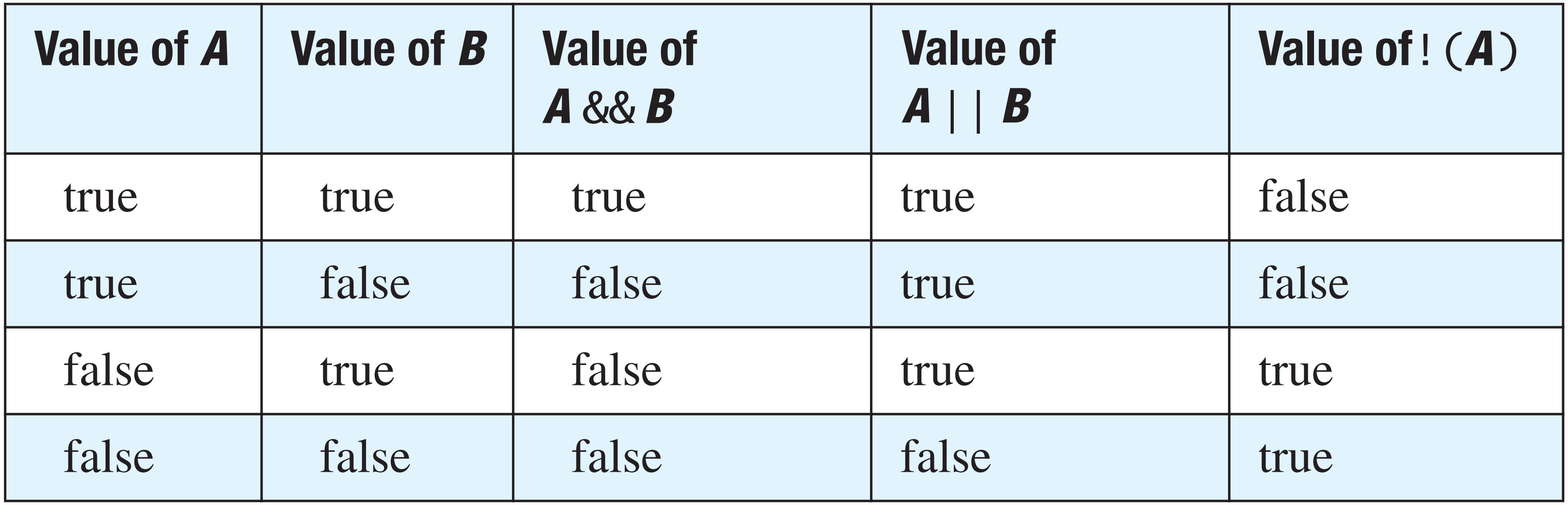
  - If the boolean expression is true, run statement 1, otherwise run statement 2

Or you can use one if statement

```
if (Boolean_Expression)
   { statements; }
Other statments
```

Boolean Expressions

- A combination of values and variables by comparison operators. Its value can only be true or false

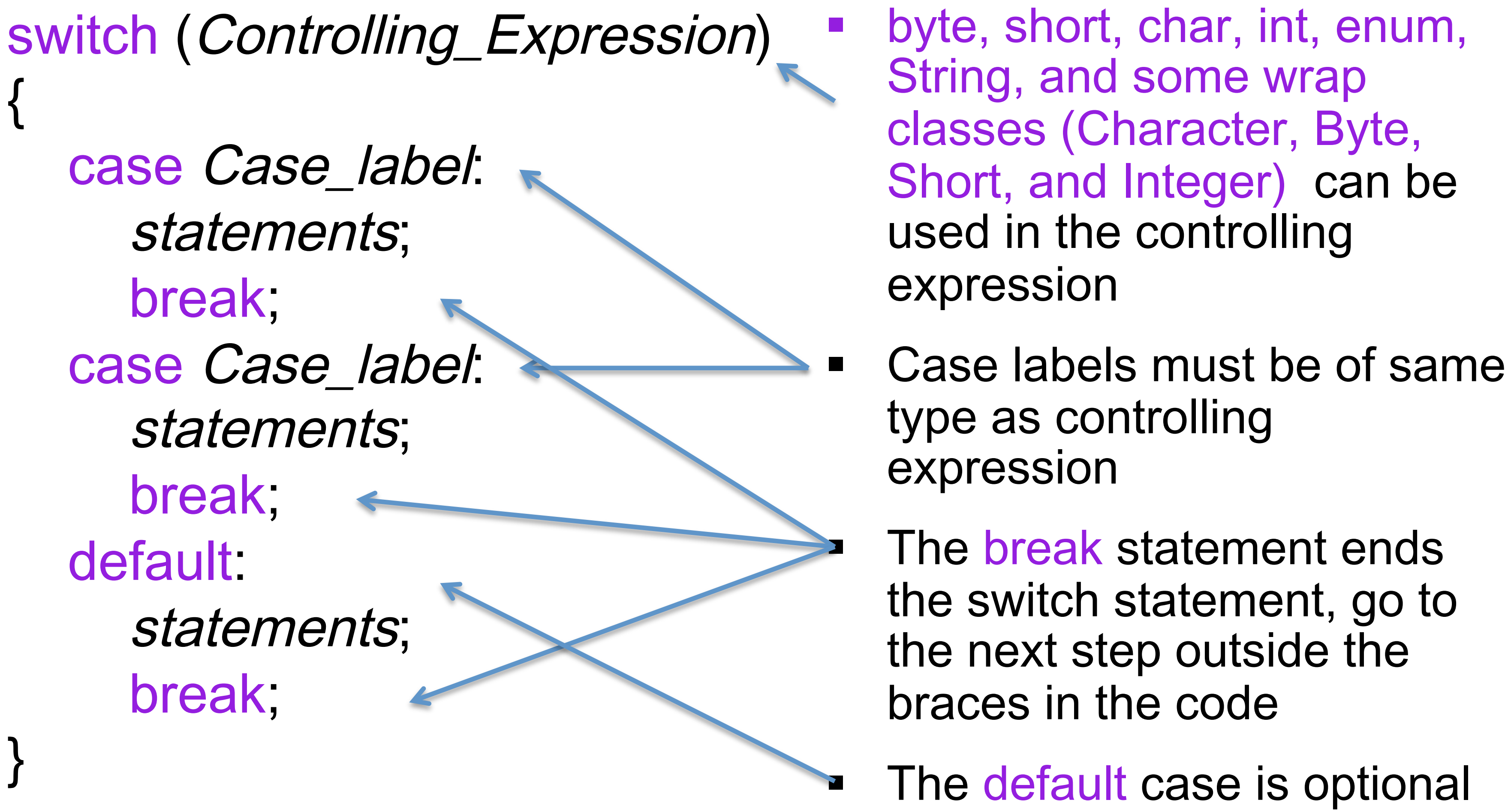


Value of *A*	Value of *B*	Value of *A* && *B*	Value of *A* || *B*	Value of ! (*A*)
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

- E.g.: int num = 6;

  boolean var = (num % 2 == 0) && (num % 3 == 0)

Branch Statements: switch



```
switch (Controlling_Expression)
{
   case Case_label:
      statements;
      break;
   case Case_label:
      statements;
      break;
   default:
      statements;
      break;
}
```

- byte, short, char, int, enum, String, and some wrap classes (Character, Byte, Short, and Integer) can be used in the controlling expression
- Case labels must be of same type as controlling expression
- The break statement ends the switch statement, go to the next step outside the braces in the code
- The default case is optional

Loop Statements

- while loop
  - Repeats its body while a boolean expression is true
- do while loop
  - Loop iterates at least ONCE
- for loop
  - Usually knows the number of iterations

Loop Statements

- Sample question:
  - Write the output for

```java
int x = 7;
boolean found = false;

do {
    System.out.print(x + " ");
    if (x <= 2)
        found = true;
    else
        x = x - 5;
} while (x > 0 && !found);
```

  - Answer: 7 2

Loop Statements

- Connection with arrays
  - E.g: Write code to declare, initialize, and fill in an array of type int, as follows

0	2	4	6	8	10	12	14	16	18

  - One way:

```
int[] a = { 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 };
```

  - Using loop:

```
int[] b = new int[10];
for (int i = 0; i < 10; i++) {
    b[i] = 2 * i;
}
```

Loop Statements

- How about an array of Class type?
  - E.g: Create an array with 5 objects of Class Student

```
Student [] arr = new Student [5];
for(int i = 0; i < arr.length; i++)
{
    arr[i] = new Student();
}
```

Loop Statements

- Nested loops
  - E.g.: Initialize each elements in a 2D array to be 30

```
int [][] table = new int[4][3];
for(int row = 0; row < table.length; row++)
{
    for(int column=0; column < table[row].length; column++)
    {
        table[row][column] = 30;
    }
}
```

Classes

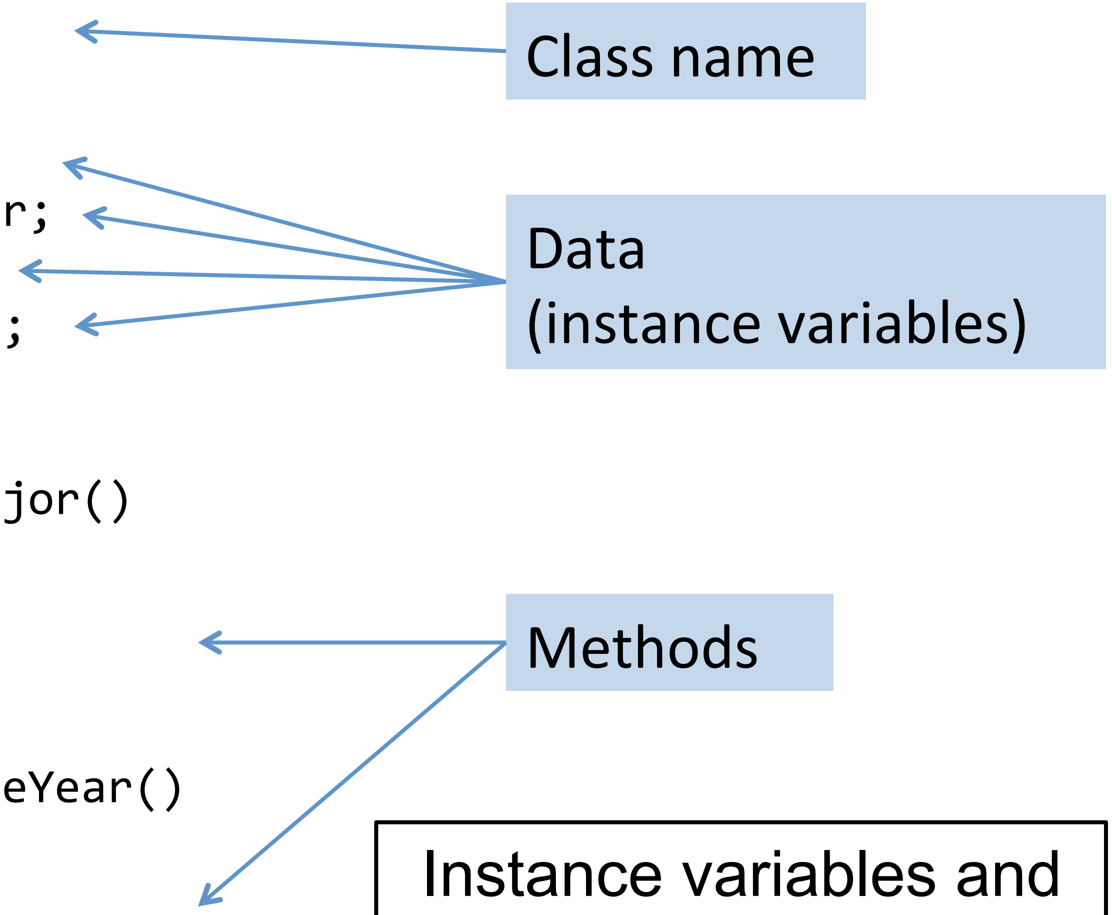
- Classes and objects
- Instance variables, local variables, and static variables
- Methods with/without return values
- Call-by-value and call-by-reference
- Public and private
- Constructors
- Static variables and methods
- Method parameters: overloading

Class and Object

- A *class* is the definition of a kind of object
  - A blueprint for constructing specific objects
- Important: classes usually do not have data; individual objects have data.
- But, a class can have variables that are static as well as methods that are static.
- Static variables and static methods belong to a class as a whole and not to an individual object

Defining a Class

```java
public class Student
{
    public String name;
    public int classYear;
    public double gpa;
    public String major;
    // ...

    public String getMajor()
    {
        return major;
    }

    public void increaseYear()
    {
        classYear++;
    }
}
```



Class name

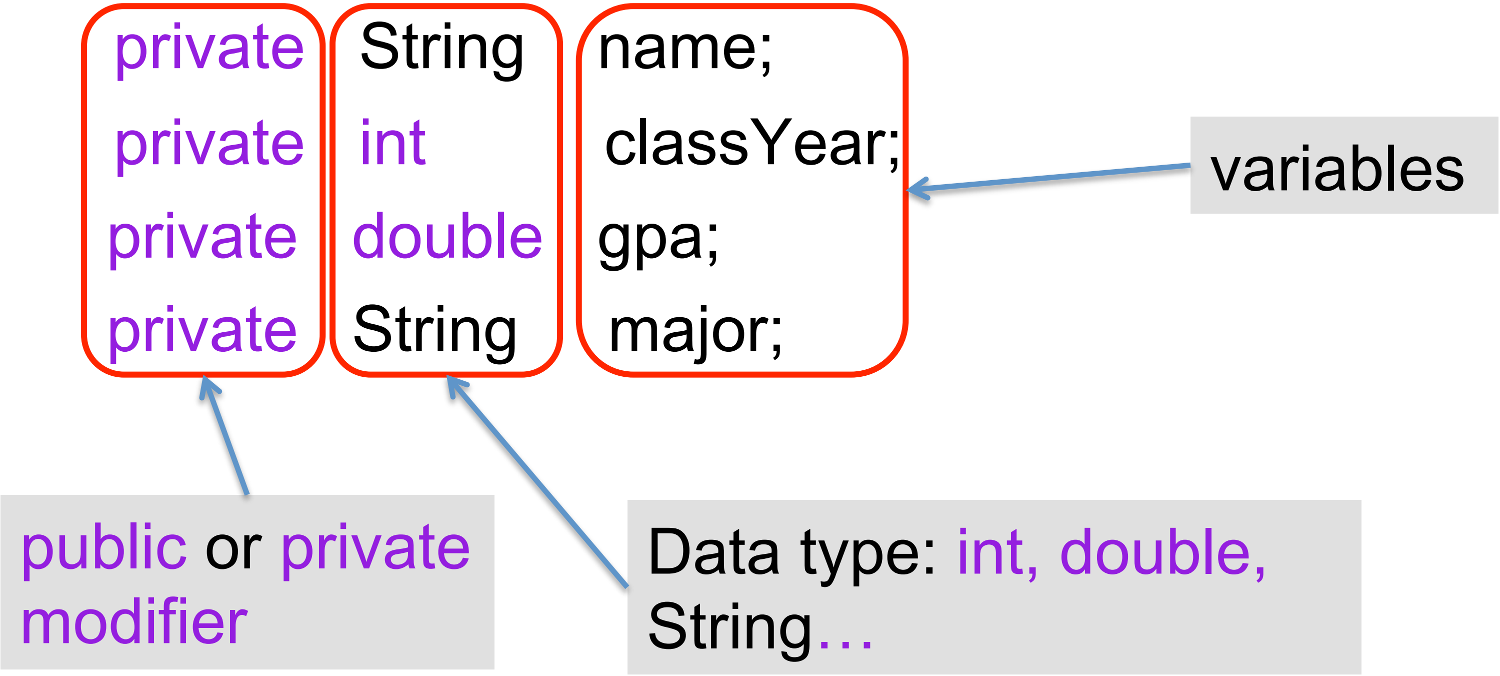
Data
(instance variables)

Methods

Instance variables and methods are members of a class

Instance Variables

- Data defined in the class are called *instance variables*



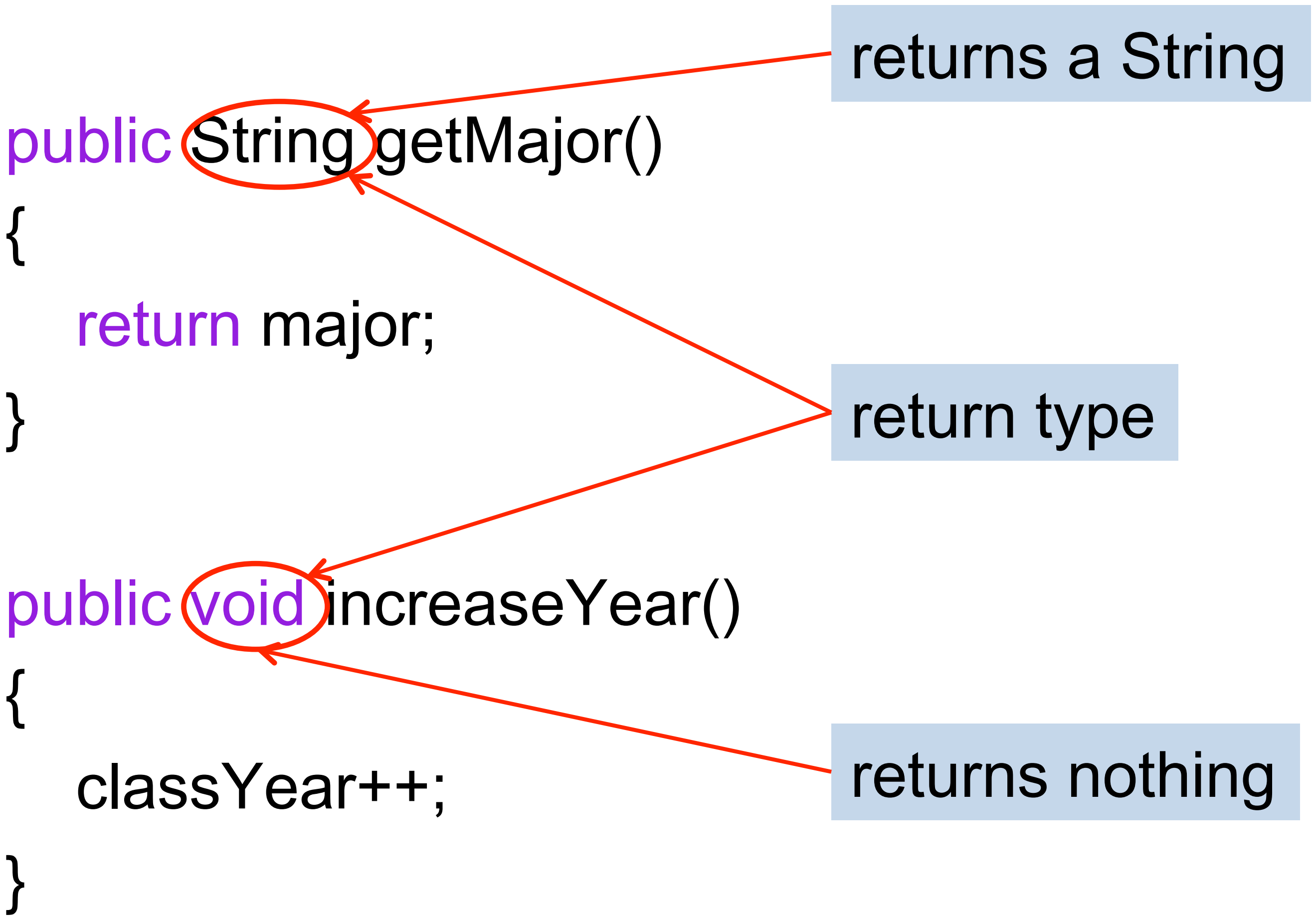
```
private String name;
private int    classYear;
private double gpa;
private String major;
```

variables

public or private modifier

Data type: int, double, String…

Methods



```
public String getMajor()
{
    return major;
}

public void increaseYear()
{
    classYear++;
}
```

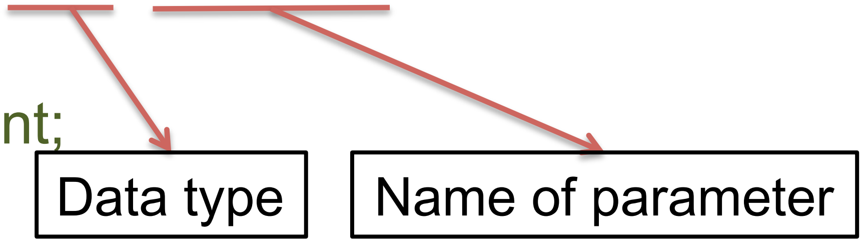
returns a String

return type

returns nothing

Method with Parameters

```
public void increaseYear(int increment)
{
    classYear += increment;
}
```



Data type

Name of parameter

```
public void increaseYear(int increment, boolean check)
{
    if (check && classYear + increment <= MaxYear ) {
          classYear += increment;
    }
}
```

- Parameters are used to hold the values that you pass to the method
- Multiple parameters are separated by comma
- The parameters are local variables

Call-by-Value

- When a method with parameter of primitive type is called:

```
public void increaseByOne( int num ) {
    num = num + 1;
}

public void doSomething () {
    int someNum = -2;
    increaseByOne( someNum );
    System.out.println( someNum );
}
```

What do you get?

Call-by-Value

- When a method with parameter of Class type is called (call-by-reference):

```
public void increaseByOne( Student s) {
        s.year = s.year + 1;
}


public void doSomething () {
        Student anna = new Student();
        anna.PID = 1234;
        anna.year = 3;
        increaseByOne( anna );
        System.out.println( anna.year );
}
```

What do you get?

public/private Modifier

- public void setMajor()
- private int classYear;

- public: there is no restriction on how you can use the method or instance variable
- private: can not directly use the method or instance variable’s name outside the class

Example

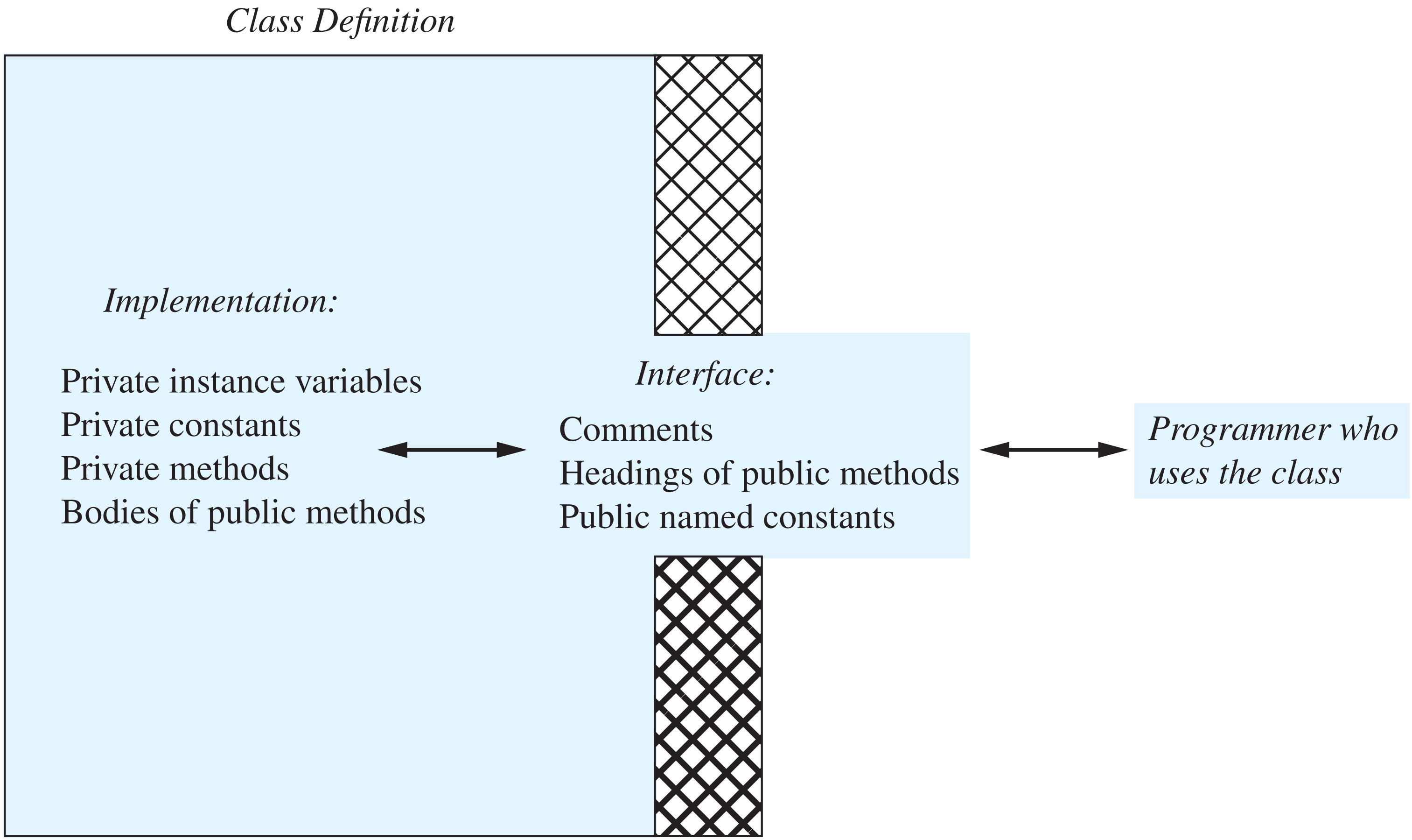
```
public class Student
{
    public int classYear;
    private String major;
}

Student jack = new Student();

jack.classYear = 1;

jack.major = “Computer Science”;
```

OK,
classYear is *public*

Error!!!
major is *private*

Information Hiding and Encapsulation

- Imagine a wall between interface and implementation



Constructors

- Constructor is a special method that is called when a new object is created

```
Student berkeley;  // not called

Student berkeley = new Student();
                        // called with new keyword
```

Constructors

- Define a constructor

```
public class Student {
      private int PID;
      private int year;
       …. Accessors & mutators …..

     public Student( int PID, int year ) {
           this.PID = PID;
           this.year = year;
     }
}
```

There is no return type or “void” keyword

Constructor has the same name as the class

If you define at least one constructor, the default constructor will not be created for you

Multiple Constructors

- You can have multiple constructors in one class
  - They all have the same name, just different parameters

```
public class Student {
    ….
    public Student( int PID, int year ) {
        this.PID = PID;
        this.year = year;
    }
    public Student( int PID ) {
        this.PID = PID;
        this.year = 1; // default case – the 1st year
    }
}
```

Default Constructor

- What if you did not write any constructor?

```
public class Student {
        private int PID;
        private int year;
        ….  No constructor …..
}

Student berkeley = new Student();
```

Java gives each class a default constructor **if you did not write any constructor**. It assigns a default value to each instance variable.

- integer, double: 0

- String and other class-type variables: null

- boolean: false

Static Members

- static variables and methods belong to a class as a whole, not to an individual object
  - One copy that all instances of the class can assess
- Static variables and methods can be accessed using the class name itself:
  - No need of an instance of the class to access it

static Version of pow Method

```
public class Math
{
    public static double PI = 3.1415926;
    // Returns x raised to the yth power, where y >= 0
    public static int pow(int x, int y)
    {
        int result = 1;
        for (int i = 0; i < y; i++)
        {
            result *= x;
        }
        return result;
    }
}
```

static keyword

```
System.out.println( Math.PI );
int z = Math.pow(2, 4);
```

static vs non-static

- All static members are at class level. They are accessed without creating any instance.
- static methods has no access to non-static members (since they belong to instances)
- Non-static methods can access both static and non-static members

Overloading

- Using the same method name for two or more methods *within the same class*
  - Example: constructors
- Parameter lists must be different
  - public double average(double n1, double n2)
  - public double average(double n1, double n2, double n3)
- Java knows what to use based on the number and types of the arguments

Method *signature*

- A method’s name and the number and types of its parameters

- signature does NOT include return type

- Cannot have two methods with the same signature in the same class

Inheritance

- What is inheritance
  - Subclasses (child/derived classes) inherit some properties from superclass (Parent/base class)
- What is overriding
  - A subclass defines a method of the same signature and the same return type as the superclass
- What is polymorphism
  - “Many forms”, each subclass object can perform its own action from overridden methods

Polymorphism and Overriding

- Dynamic binding

```
public class Animal {
    private String animalName;
    public void speak() {
    // default method -- can be empty
    }

    public static void main(String[] args)
    {
        Animal a[] = new Animal[3];
        a[0] = new Cat();
        a[1] = new Dog();
        a[2] = new Duck();
        for (int i = 0; i < 3; i++) {
            a[i].speak();
        }
    }
}
```

```
public class Cat extends Animal {
    public void speak() {
        System.out.println("MEW");
    }
}

public class Dog extends Animal {
    public void speak() {
        System.out.println("WOOF");
    }
}

public class Duck extends Animal {
    public void speak() {
        System.out.println("QUACK");
    }
}
```

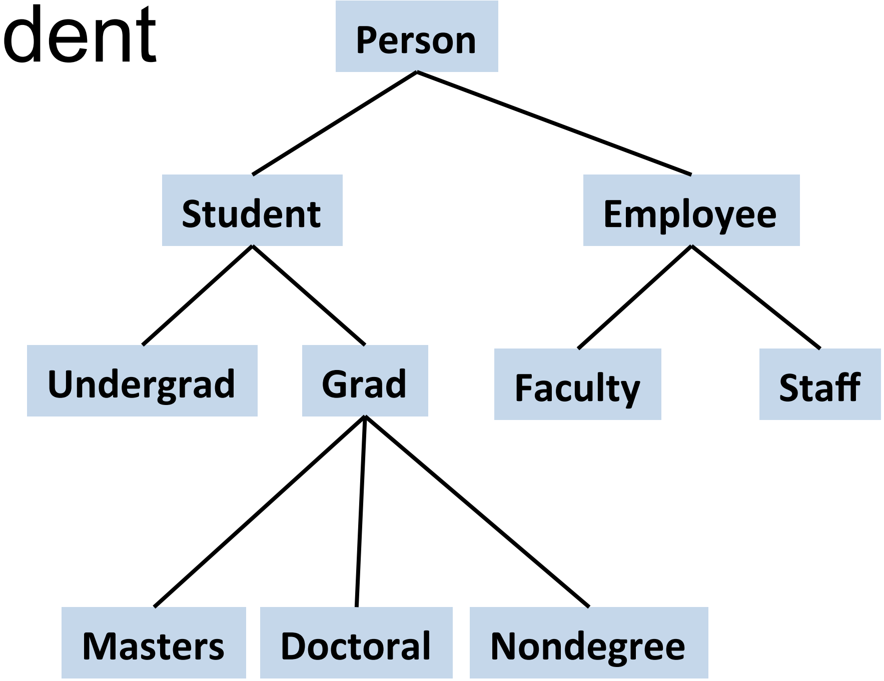
Output: MEW, WOOF, QUACK

The *is-a* Relationship

- This inheritance relationship is known as an *is-a relationship*

- A Doctoral student ***is a*** Grad student
- A Grad student ***is a*** Student
- A Student ***is a*** Person

- Is a Person a Student?
  - Not necessarily!



Type Compatibilities

- `Person per1 = new Person();`
- `Student std1 = new Student();`
- `Person per2 = std1;`
  - Yes! A student is a person
- `Student Std2 = Per1;`
  - No! A person is not necessarily a student

Creating an Array

```
int[] scores = new int[5];
```

- This is like declaring 5 strangely named variables of type int:
  - `scores[0], scores[1], scores[2], scores[3], scores[4]`
- The base type can be any type

```
double[] temperature = new double[7];
Student[] students = new Student[35];
```

- Indices MUST be within bounds
  - `Temperature[7] = 0.0; //ERROR! Index out of bounds`

Finding the Length of an Existing Array

- An array is a special kind of object
  - It has one public instance variable: ***length***
  - ***length*** is equal to the length of the array

    ```
    Pet[] pets = new Pet[20];
    pets.length has the value 20
    ```

  - You cannot change the value of *length*
  - *Once declared, an array cannot be resized!*

Arrays as Instance Variables

```
public class Weather
{
    private double[] temperature;
    private double[] pressure;

    public void initializeTemperature(int len)
    {
        temperature = new double[len];
    }
}
```

Arrays as Parameters

```
public void changeArray(int[] arr)
{
    int len = arr.length;
    arr[len – 1] = 25;
}
```

23	47	52	14	25

Arrays as Return Types

```
public double[] buildArray(int len)
{
    double[] retArray = new double[len];
    for (int i = 0; i < retArray.length; i++)
    {
        retArray[i] = i * 1.5;
    }

    return retArray;
}
```

Declaring and Creating 2D Arrays

```
int[][] table = new int[4][3];
```

or

```
int[][] table;
table = new int[4][3];
```

How do you use a 2D array?

- How about a 2D array?

```
int[][] table = new int[4][3];
```

- Use a nested loop

```
for(int row = 0; row < table.length; row++)
{
    for(int column=0; column < table[row].length; column++)
    {
        table[row][column] = 30;
    }
}
```

2D Array of Irregular Shape

```
int[][] x = new int[3][];
x[0] = new int[1];
x[1] = new int[2];
X[2] = new int[3];

System.out.println(x[0].length);
System.out.println(x[1].length);
System.out.println(x[2].length);
```

Array and ArrayList

- Array: fixed size. Good if the size is known and fixed
  - my1DArray[index], my2DArray[i][j] : use as variables
  - my1DArray.length, my2Darray[i].length : this is a public instance variable. Not a method
  - 1D, 2D, … n-D, 2D array does not have to be rectangle

- ArrayList: dynamic size. Use methods to manipulate data
  - add, get, set, size, remove …..
  - Only stores objects. Need wrapper class for primitive values

Example: ArrayList

-
  ```
   //ArrayList to Store only String objects
  ArrayList<String> stringList
                = new ArrayList<String>();
  ```
- `stringList.add("Item");`
- `String item = stringList.get(i);`
- `int size = stringList.size();`
- `boolean result = stringList.isEmpty();`
- `int index = stringList.indexOf("Item");`

-
  ```
  stringList.remove(item); or
  stringList.remove(0);
  ```

Recursion

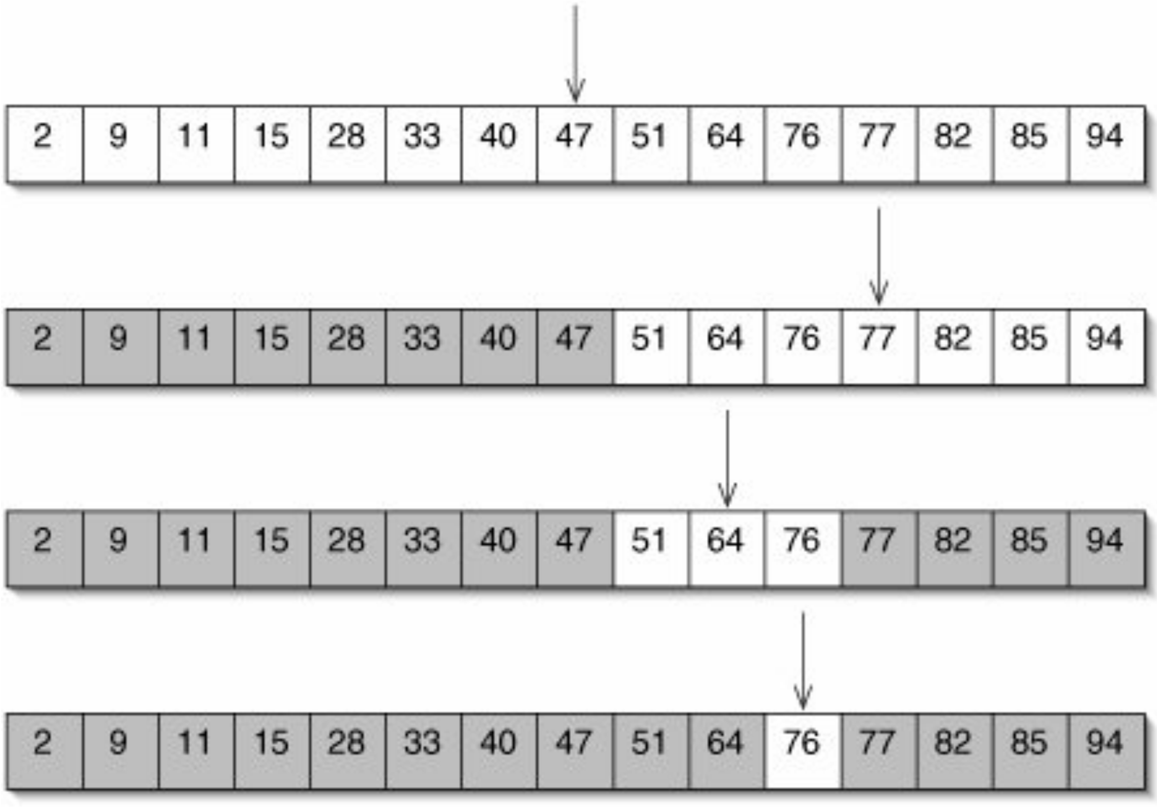
- Recursive: an algorithm has one subtask that is a smaller version of the entire algorithm's task
- Recursion: you write a method to solve a big task, and the method invokes itself to solve a smaller subtask
- Base case: the smallest task
- Recursive rule: relationship between the big task and its subtasks

Sequential (Linear) Search

- Basic idea
  - For each item in the list:
    - if that item has the desired value, stop the search and return the item's location.
  - Return *Not Found*.
- No faster algorithm for unsorted array
- For sorted array, we can use binary search

Binary Search

- Works for sorted array, reduces half searching space in each iteration



2 9 11 15 28 33 40 47 51 64 76 77 82 85 94

2 9 11 15 28 33 40 47 51 64 76 77 82 85 94

2 9 11 15 28 33 40 47 51 64 76 77 82 85 94

2 9 11 15 28 33 40 47 51 64 76 77 82 85 94

Selection

- One selection problem:
  - Find the smallest / largest number in a given list (array)
  - No assumption made on the list (so it is not sorted)

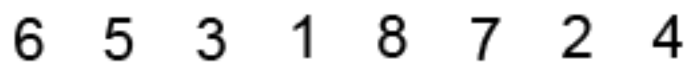
Sorting

- Bubble sort
- Selection sort
- Merge sort

Bubble Sort (or Sinking Sort)

- Basic idea (Wikipedia)
  - Start from the beginning of the list
  - Compare every adjacent pair, swap their positions if they are not in the right order
  - After each iteration, one less element (the last one) is needed to be compared until there is no more elements left to be compared

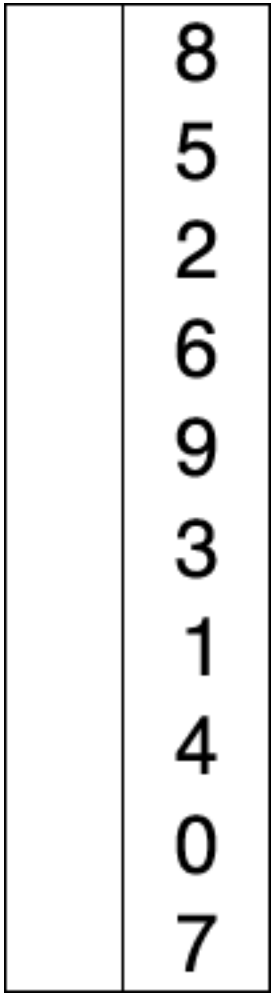
Animation from Wikipedia:



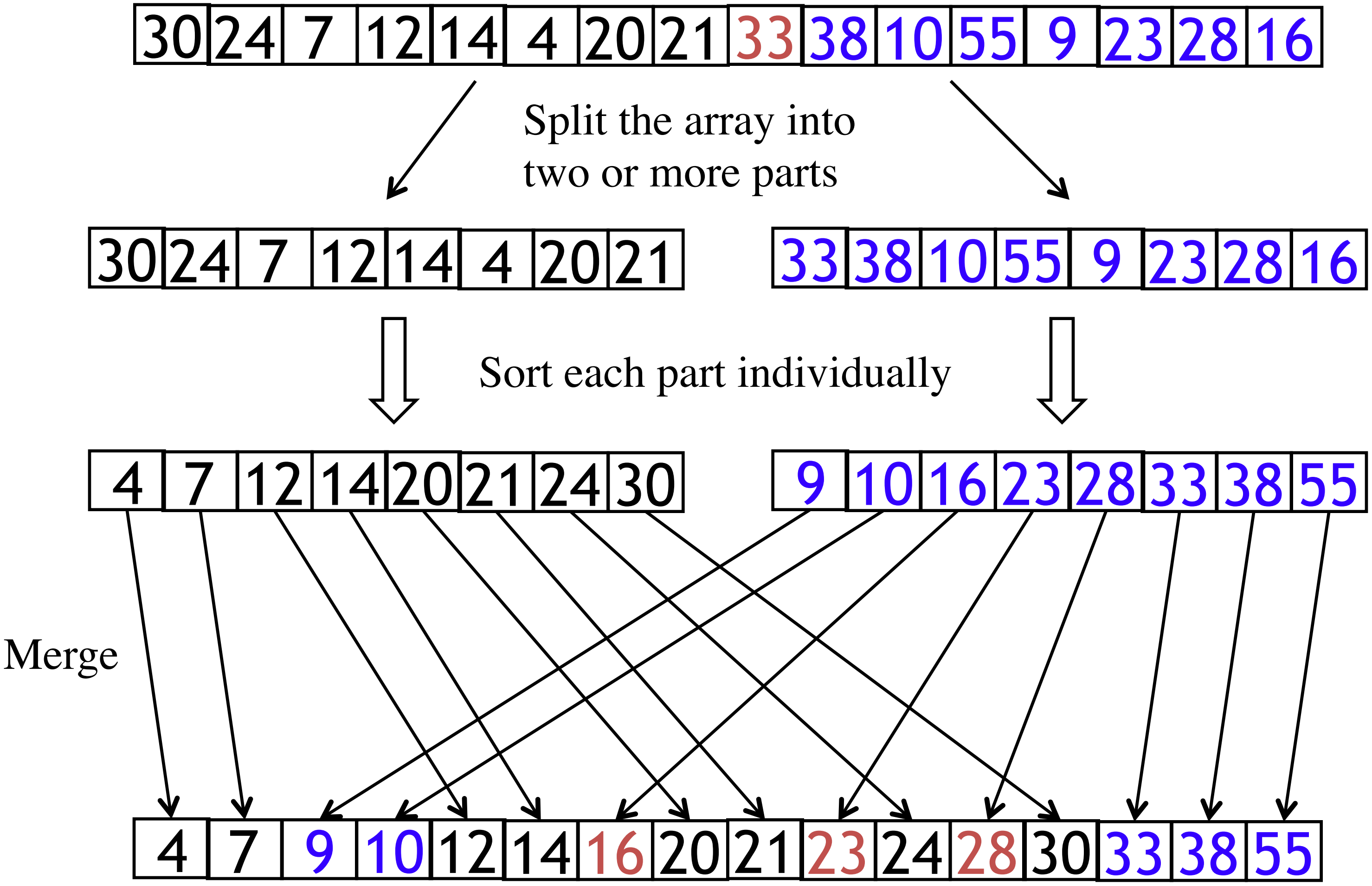
Selection Sort

- Given an array of length n, each time select the smallest one among the rest elements:
  - Search elements 0 through n-1 and select the smallest
    - Swap it with the element at location 0
  - Search elements 1 through n-1 and select the smallest
    - Swap it with the element at location 1
  - Search elements 2 through n-1 and select the smallest
    - Swap it with the element at location 2
  - Search elements 3 through n-1 and select the smallest
    - Swap it with the element at location 3
  - Continue until there’s no element left

Animation from Wikipedia:



Merge Sort



30 24 7 12 14 4 20 21 33 38 10 55 9 23 28 16

Split the array into two or more parts

30 24 7 12 14 4 20 21

33 38 10 55 9 23 28 16

Sort each part individually

4 7 12 14 20 21 24 30

9 10 16 23 28 33 38 55

Merge

4 7 9 10 12 14 16 20 21 23 24 28 30 33 38 55

Exception Handling

- Try-throw-catch
  - **Try block**: detects exceptions
  - **Throw an exception**: report a problem and asks for some code to handle it properly
  - **Catch block: catches an exception**, a piece of code dedicated to handle one or more specific types of problem

Creating a Text File

- Opening a file connects it to a stream
- The class PrintWriter in the package java.io is for writing to a text file

```
String fileName = "out.txt";//Could read file name from user
PrintWriter outputStream = null;
try
{
    outputStream = new PrintWriter(fileName);
}
catch(FileNotFoundException e)
{
    System.out.println("Error opening the file " + fileName);
    System.exit(0);
}
```

Creating a Text File

- After we connect the file to the stream, we can write data to it
  - outputStream.println(“This is line 1.”);
  - outputStream.println(“Here is line 2.”);
- Closing a file disconnects it from a stream
  - outputStream.close();

Reading From a Text File

- Use Scanner to open a text file for input

```
Scanner Stream_Name = new Scanner(new File(File_Name));
```

  - E.g.: Scanner inputStream = new Scanner(new File(“out.txt”));

- Use the methods of Scanner to read

```
while (inputStream.hasNextLine())
{
    String line = inputStream.nextLine();
    System.out.println(line);
}
```

Thank you !!!