

CSCI 132:

Basic Data Structures and Algorithms

Midterm Review

Reese Pearsall
Spring 2023

Announcements

No Lab tomorrow

Midterm Exam Wednesday

Program 2 due **Friday** 3/10 @ 11:59

Declaring Variables

Primitive Data Types

- `int`
- `double`
- `boolean`
- `char`
- `float`

Non-Primitive Data Types

- `String`

```
String s = "Reese";  
String last_name = "Pearsall";  
System.out.println(s + last_name)
```

Valid Variable Declaration

```
int i = 5;  
int x;  
int num = 125;  
  
char grade = "A";  
  
boolean flag = true;
```

When we declare a variable, we **must** define the datatype as well

Invalid Variable Declaration

Operators

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulo)
- + (String concatenation)
- ++ (Increment)
- -- (Decrement)

```
int x, y, answer;  
x= 2;  
y = 3;  
answer = x + y;
```

Using the plus operator (+) between two values that are Strings will result in **String concatenation**

```
String x = "hi ";  
String y = "there";  
System.out.println(x + y);
```

```
>> hi there
```

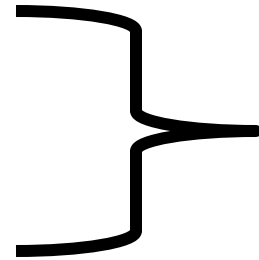
Increment operator (++) will add 1 to a variable

```
int counter = 0;  
System.out.println(counter);  
counter++;  
System.out.println(counter);  
counter++;  
System.out.println(counter);
```

```
>> 0  
    1  
    2
```

```
public class Student {
```

```
    private String name;  
    private String major;  
    private int num_of_credits;  
    private double gpa;  
    private String year;
```



Instance fields of our Student Class

private means they can not be directly accessed outside of the class

```
    public Student(String name, String major, int num_of_credits, double gpa) {  
        this.name = name;  
        this.major = major;  
        this.num_of_credits = num_of_credits;  
        this.gpa = gpa;  
        this.year = "Unknown";  
    }
```

Student.Java

```
public class StudentDemo {
```

```
    public static void main(String[] args) {
```

```
        Student student1 = new Student("Charles", "Computer Science", 75, 3.5);
```

StudentDemo.Java

```
public class Student {  
  
    private String name;  
    private String major;  
    private int num_of_credits;  
    private double gpa;  
    private String year;
```

This is the **constructor**, the special method that creates our objects
Each of our “blueprints” needs a constructor

```
    public Student(String name, String major, int num_of_credits, double gpa) {  
        this.name = name;  
        this.major = major;  
        this.num_of_credits = num_of_credits;  
        this.gpa = gpa;  
        this.year = "Unknown";  
    }
```

Student.Java

```
public class StudentDemo {  
  
    public static void main(String[] args) {
```

```
        Student student1 = new Student("Charles", "Computer Science", 75, 3.5);
```

StudentDemo.Java

```
public class Student {  
  
    private String name;  
    private String major;  
    private int num_of_credits;  
    private double gpa;  
    private String year;  
  
    public Student(String name, String major, int num_of_credits, double gpa) {  
        this.name = name;  
        this.major = major;  
        this.num_of_credits = num_of_credits;  
        this.gpa = gpa;  
        this.year = "Unknown";  
    }  
}
```

Student.Java

```
public class StudentDemo {  
  
    public static void main(String[] args) {  
  

```

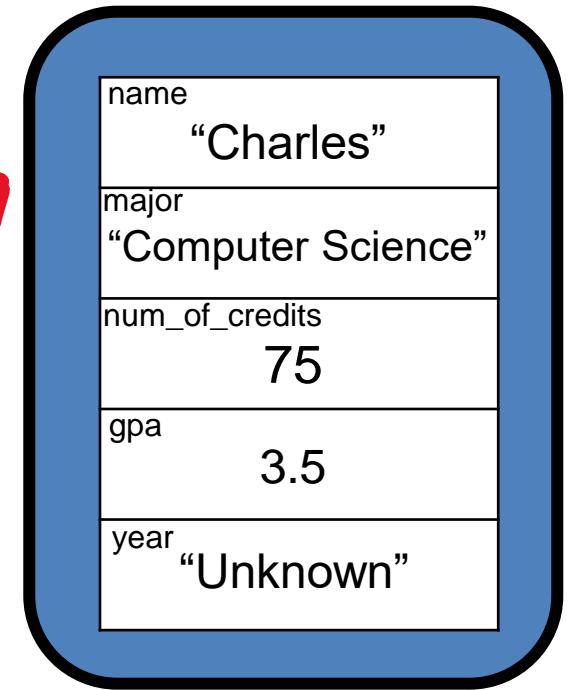
When we use the **new** keyword, it will invoke our constructor

```
        Student student1 = new Student("Charles", "Computer Science", 75, 3.5);  
    }  
}
```

StudentDemo.Java

```
public class Student {  
  
    private String name;  
    private String major;  
    private int num_of_credits;  
    private double gpa;  
    private String year;  
  
    public Student(String name, String major, int num_of_credits, double gpa) {  
        this.name = name;  
        this.major = major;  
        this.num_of_credits = num_of_credits;  
        this.gpa = gpa;  
        this.year = "Unknown";  
    }  
}
```

student1



Student.Java

```
public class StudentDemo {  
  
    public static void main(String[] args) {  
  
        Student student1 = new Student("Charles", "Computer Science", 75, 3.5);  
    }  
}
```

StudentDemo.Java


```
public class Student {
```

(instance fields and constructor go here)

```
    public String getName() {  
        return this.name;  
    }
```

Name of method

This method returns a String

This method is public (other classes can use it)

The **this** keyword refers to the *object* that this method was called on (student1)
(return student1's name attribute)

Student.Java

```
public class StudentDemo {
```

```
    public static void main(String[] args) {
```

```
        Student student1 = new Student("Charles", "Computer Science", 75, 3.5);
```

```
        System.out.println(student1.getName());
```

StudentDemo.Java

Example: A student is allowed to register for CSCI 476 if they have a GPA greater than 2.0, **and** if they are a Junior **or** Senior

```
public void allowToRegister() {  
  
    if (this.gpa > 2.0) { // check the first condition (Alternatively, we could use an && here)  
  
        if (this.year.equals("Junior") || this.year.equals("Senior")){  
  
            System.out.println("Student is allowed to register for CSCI 476");  
  
        }  
  
    }  
  
}
```

We can check one of two conditions is true using the or operator (||)

Student.Java

(we do not have the **or** keyword in Java)

```
student1.allowToRegister();
```

StudentDemo.Java

```
public void determineYear() {  
    if(this.num_of_credits <= 30) {  
        this.year = "Freshman";  
    }  
    else if(this.num_of_credits > 30 && this.num_of_credits <= 60) {  
        this.year = "Sophomore";  
    }  
    else if(this.num_of_credits > 60 && this.num_of_credits <= 90) {  
        this.year = "Junior";  
    }  
    else if(this.num_of_credits > 90 && this.num_of_credits <= 120) {  
        this.year = "Senior";  
    }  
    else {  
        this.year = "???";  
    }  
}
```

We can check multiple conditions using the and operator (&&)

(we do not have the **and** keyword in Java)

Student.Java

```
student1.determineYear();
```

StudentDemo.Java

Arrays are a *collection* of data
→ Once initialized, are **fixed** in size
→ Can only hold one data type



```
System.out.println(test_scores[2]);  
>> 65  
  
System.out.println(test_scores[4]);  
.
```

Declaring an array and giving it a value

```
int[] test_scores = {99, 81, 65, 46};
```

Declaring an array allocating 5 empty spots (we need to fill them later)

```
String[] names = new String[5];
```

test_scores	0	1	2	3	
	99	81	65	46	
names	0	1	2	3	4
	null	null	null	null	null

For loops can be used to iterate across an array.

Two ways:

1. Iterate by index

```
String[] animals = {"Zebra", "Elephant", "Lion", "Penguin"};

for (int i = 0; i < animals.length; i++) {

    System.out.println(animals[i]);

}
```

2. Iterate by element

```
for (String i : animals) {
    System.out.println(i);
}
```

Both will give you the
exact same output...

While loops can be used to iterate if a condition is true.

```
int x = 100;
while(x > 0) {

    System.out.println(x);
    x--;

}
```

1. Check Condition
2. If condition is true, execute body of loop
3. Repeat

```
int x = 100;
while(x > 0) {

    System.out.println(x);
    x++;

}
```

You do have to worry about the possibility of infinite loops....

New

The **do/while** loop will always execute the body of the loop once, and then check the condition

```
int i = 0;
do {
    System.out.println(i);
    i++;
}
while (i < 5);
```

1. Execute body of loop
2. Check condition
3. Repeat

!!! You are guaranteed at least one execution of the loop body

```

public class ReferencesDemo {
    public static void main(String[] args) {

        Person person1 = new Person("Jim Bob", 44);
        Person person2 = new Person("Sally", 28);

        Person person3 = person1;

    }
}

```

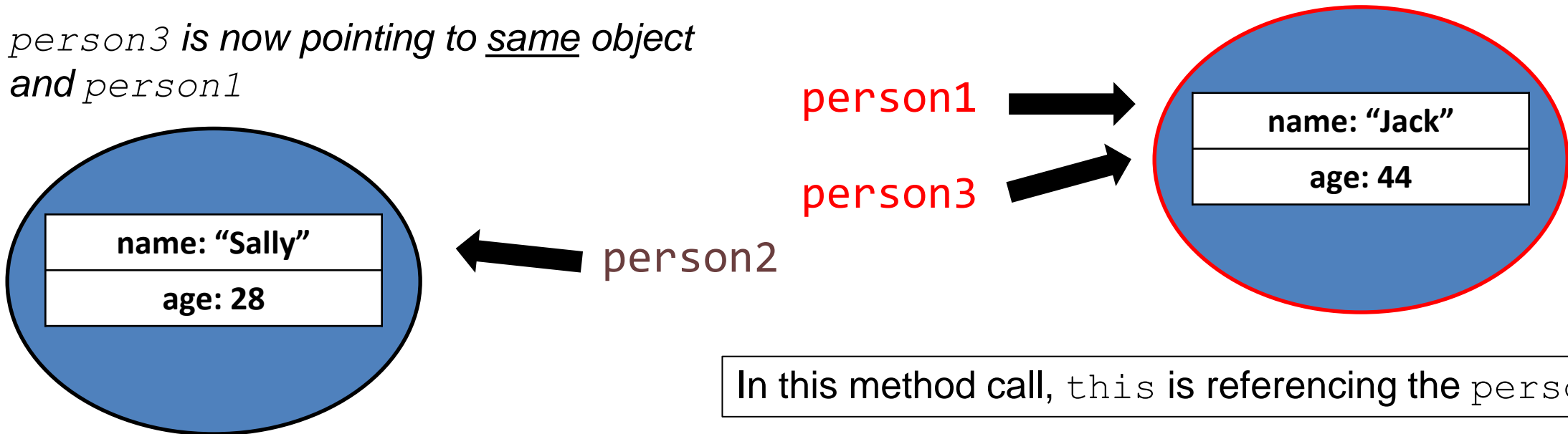
```

public void changeName(String newName) {
    this.name = newName;
}

```

Suppose we create a new reference variable and link it to an existing object

person3 is now pointing to same object and person1



In this method call, `this` is referencing the `person1` object


```

public class ReferencesDemo {
    public static void main(String[] args) {

        Person person1 = new Person("Jim Bob", 44);
        Person person2 = new Person("Sally", 28);

        Person person3 = person1;
        person1.changeName("test");

    }
}

```

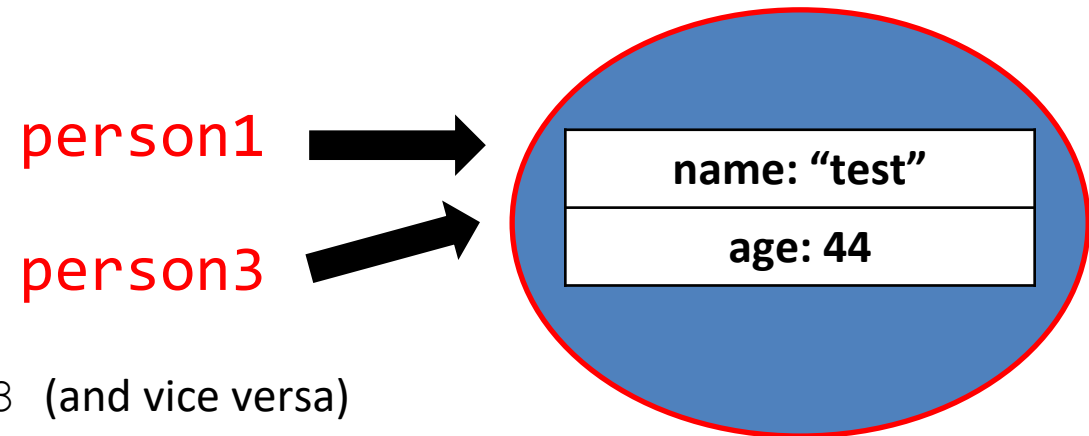
```

public void changeName(String newName) {
    this.name = newName;
}

```

Suppose we create a new reference variable and link it to an existing object

person3 is now pointing to same object and person1



Any changes to person1 will also update person3 (and vice versa)

System.out.println(person1.getName()) → "test"

System.out.println(person3.getName()) → "test"

Inheritance is a mechanism in Java that allows for a class to acquire instance fields and methods from another class

In Java, we use the **extends** keyword to indicate that a class is inheriting from another

```
public class Programmer extends Employee {  
}
```

The `Programmer` class inherits from the `Employee` class

```

public class Programmer extends Employee {

    private String programming_language;

    public Programmer(String name, int id, int salary, String lan) {
        super(name, id, salary);
        this.programming_language = lan;
    }

    public String getLanguage() {
        return this.programming_language;
    }

}

```

Programmer.java

```

public class Employee {

    private String name;
    private int emp_id;
    private int salary;

    public Employee(String name, int id, int salary) {
        this.name = name;
        this.emp_id = id;
        this.salary = salary;
    }

    public String getName() {
        return this.name;
    }

}

```

Inherited!

Employee.java

```

Programmer reese = new Programmer("Reese Pearsall", 1234, 90000, "Python");
System.out.println(reese.getName());

```

`getName()` is not defined in the `Programmer` class, but because the `Programmer` class *inherits* from the `Employee` class, the `reese` object has access to the `getName()` method

```

public class Programmer extends Employee {

    private String programming_language;

    public Programmer(String name, int id, int salary, String lan) {
        super(name, id, salary);
        this.programming_language = lan;
    }

    public String getLanguage() {
        return this.programming_language;
    }

}

```

Programmer.java

```

public class Employee {

```

```

    private String name;
    private int emp_id;
    private int salary;

```

```

✓ public Employee(String name, int id, int salary) {
    this.name = name;
    this.emp_id = id;
    this.salary = salary;
}

public String getName() {
    return this.name;
}

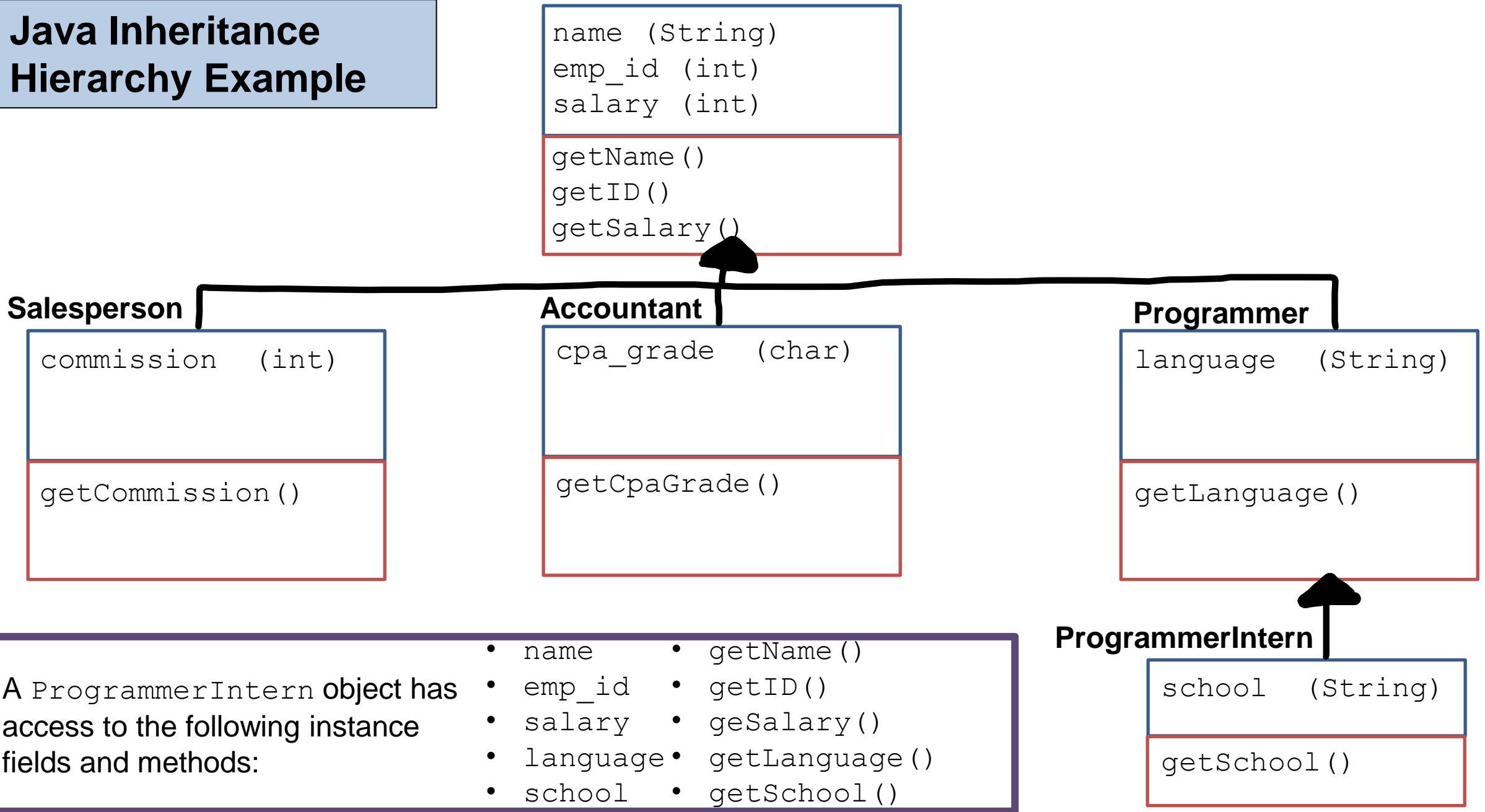
```

Employee.java

private instance fields and methods are **not** inherited

Instead, we can use the `protected` keyword

Java Inheritance Hierarchy Example



Static methods are methods in Java that can be called without creating an object of a class

```
public class StaticDemo {  
    public static void main(String[] args) {  
        AnotherClass.funMethod("Hello");  
    }  
}
```

StaticDemo.java

```
public class AnotherClass {  
    public static void funMethod(String arg)  
    {  
        System.out.println(arg);  
    }  
}
```

AnotherClass.java


If the static method is in another class, we can access it by giving the class name (`AnotherClass`)

Once again, I do not need to create an `AnotherClass` object to call this static method


However, now objects are no longer an implicit argument to this method (cant use `this` anymore)

Abstract Classes are restricted classes that cannot be used to create objects. To access it, it must be inherited from another class.

```
public abstract class Employee {  
    ...  
}
```

 `Employee e = new Employee("Sally", 4444, 123456);`

You **cannot** create instances of an abstract class.

`Accountant kevin = new Accountant("Kevin Malone", 4444, 42000, 'C');` 

Instead, we use objects from another class *that inherits from the abstract class*

Interfaces are abstract classes that only contain methods with no body

```
public interface Vehicle {  
    void accelerate(int a);  
    void slowdown(int a);  
    void refuel(int a);  
}
```

Now, any Class that also has the behavior of `accelerating`, `slowdown`, and `refuel` can implement our interface, and those classes are **forced** to write the body of the methods

```
public class Ferrari implements Vehicle {  
  
    @Override  
    public void accelerate(int a) {  
        ...  
    }  
  
    @Override  
    public void slowdown(int a) {  
        ...  
    }  
  
    @Override  
    public void refuel(int a) {  
        ...  
    }  
}
```

The code of the method body is omitted, but that is where the programmer can put the specific behavior of:

- how a Ferrari will accelerate
- how a Ferrari will slow down
- how a Ferrari will refuel

Interfaces are abstract classes that only contain methods with no body

Why use interfaces?

Interfaces are great when you need **multiple implementations** of the **same behavior**

It forces classes to implement X methods that might not logically belong to them (*more control*)

It provides **abstraction**
(ie the details of how things are implemented are not revealed in an interface)

Inheriting from a class

Class inherits **instance fields** and **methods**

Can only inherit from one class

Sub class is **not required** to override methods

Implementing an Interface

Class inherits **methods** with no bodies

Can implement multiple interfaces

Sub class is **required** to override methods

Polymorphism is the ability of a class to provide different implementations of a method, depending on the *type of object* that is passed to the method.

```
Bird a2 = new Bird("Puffin",27.0, "North America",7400000,21.5);  
Wolf b2 = new Wolf("Arctic Wolf",120.0, "North America",200000, 16);  
  
a2.makeSound();  
b2.makeSound();
```

The `makeSound()` method does something different for each object

Array Limitations

Cons

- **Can't change the length** *What can we do about this?*
- Can only store one data type

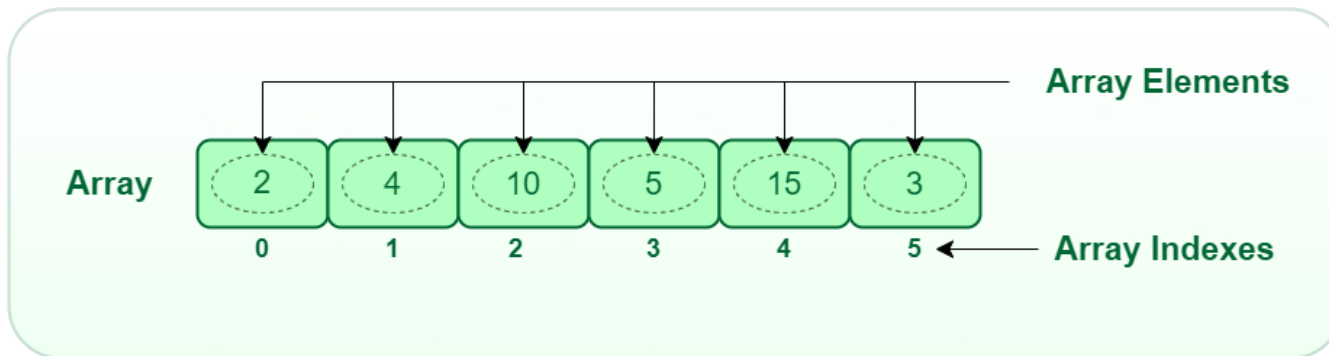
```
int[] myArray = {1, 2, 3};  
System.out.println(Arrays.toString(myArray));
```

```
int[] newArray = new int[myArray.length + 1];           // Create a new array that is one spot bigger  
for(int i = 0; i < myArray.length; i++) {  
    newArray[i] = myArray[i];                             // Fill new array with contents of old array  
}
```

```
int new_value = 4;  
newArray[myArray.length] = new_value;                   // add new value to array  
myArray = newArray;                                     // Update reference variable
```

An **ArrayList** is a data structure that can hold multiple, similar values (just like an array), **BUT**

- Dynamic, can easily resize
- Can easily add new elements and remove elements
- Like a Python list 😊



Somebody took `arrays`, and made them better

- Still have indices
- Still can only store one data type

Java ArrayLists

We first need to remember to import it ☺

```
import java.util.ArrayList;
```

Creating a new ArrayList

```
ArrayList<String> mylist = new ArrayList<String>();
```

We can add stuff to the ArrayList using the `.add()` method (built in method!)

```
mylist.add("Jack");
```

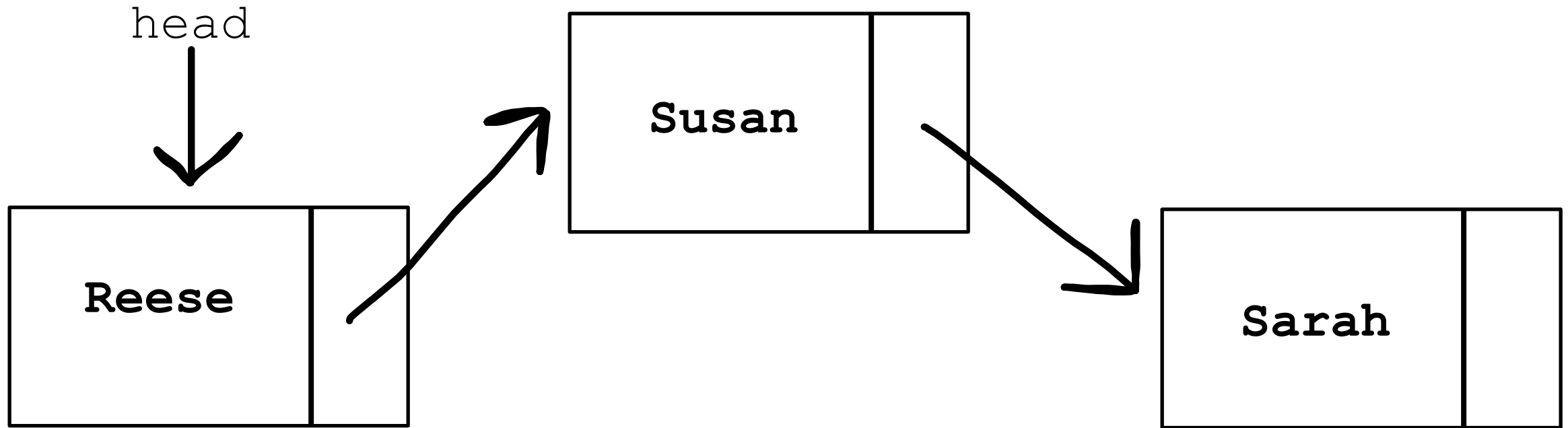
To access elements in the array, we use the `.get()` method (we cannot use the square bracket index `[]`)

```
System.out.println(mylist.get(2)); // this will print the String at index 2
```

We can remove stuff by index, or by searching for a specific element

```
mylist.remove("Eli");  
mylist.remove(0);
```

A **Linked List** is a data structure that consists of a collection of connected nodes



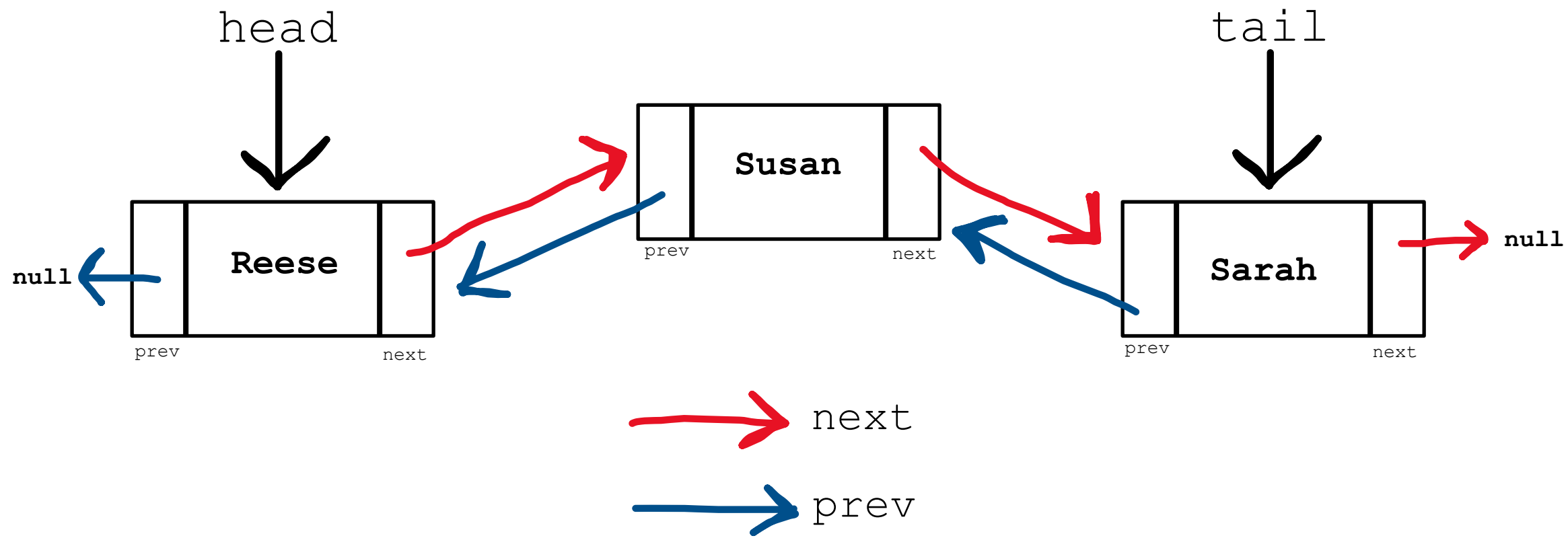
Nodes consists of **data** (String, int, array, etc) and a **pointer to the next node**

A Linked List also has a pointer to the start of the Linked List (`head`)

Singly Linked List Methods (No `tail`)

- `addToFront()` - adds new node to beginning of LL $O(1)$
- `addToBack()` – adds new node to end of LL $O(N)$
- `removeFirst()` – removes first node of LL $O(1)$
- `removeLast()` – removes last node of LL $O(N)$
- `printLinkedList()` – prints nodes and their data $O(N)$

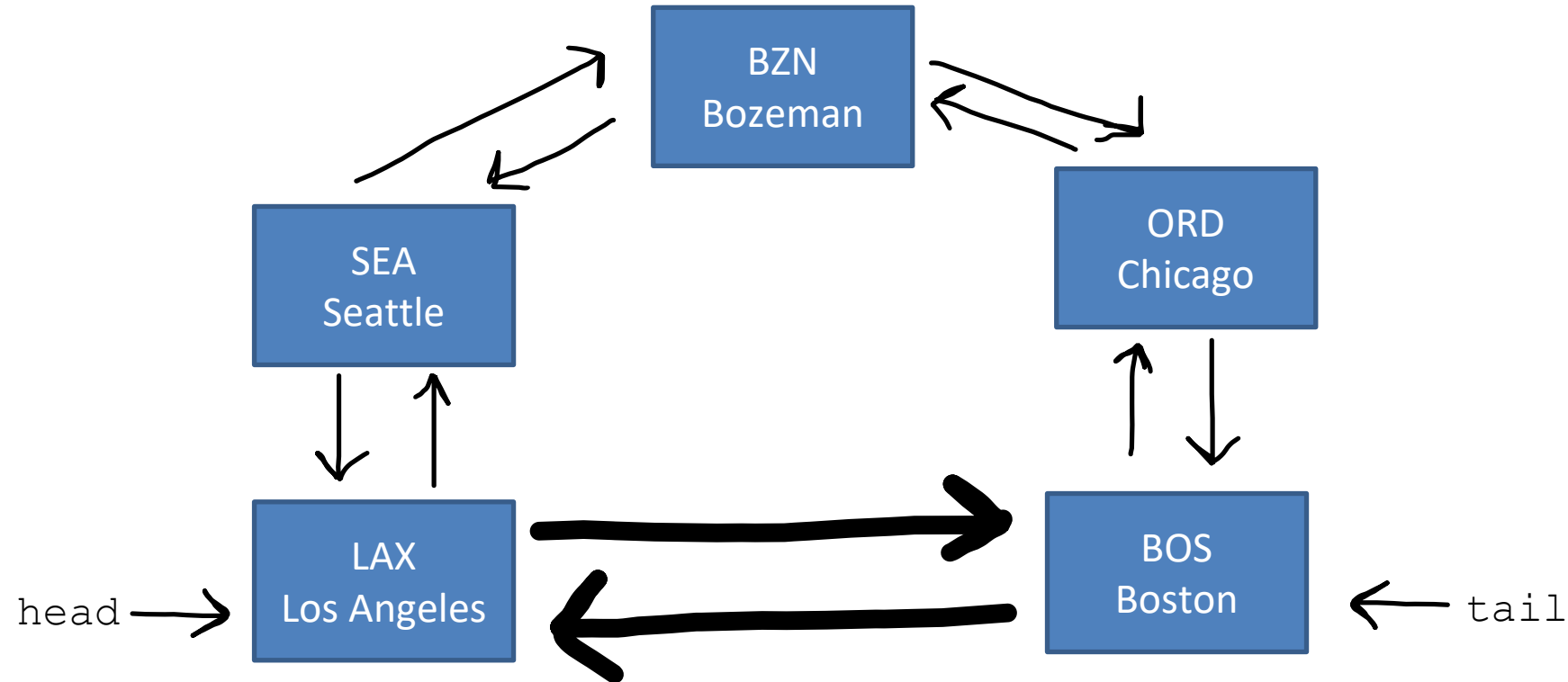
A **Doubly Linked List** keeps track of the next node and the previous node



Doubly Linked List Methods

- `addToFront()` - adds new node to beginning of LL $O(1)$
- `addToBack()` – adds new node to end of LL $O(1)$
- `removeFirst()` – removes first node of LL $O(1)$
- `removeLast()` – removes last node of LL $O(1)$
- `printLinkedList()` – prints nodes and their data $O(N)$
- `insert(N)` – insert node at spot N $O(N)$

A **Circular Linked List** is a linked list where the first and last node are connected, which creates a circle



Circular Doubly Linked List Methods

- `addToFront()` - adds new node to beginning of LL $O(1)$
- `addToBack()` – adds new node to end of LL $O(1)$
- `removeFirst()` – removes head node of LL $O(1)$
- `removeLast()` – removes tail node of LL $O(1)$
- `printLinkedList()` – prints nodes and their data $O(N)$
- `insert(N)` – insert node at spot N $O(N)$

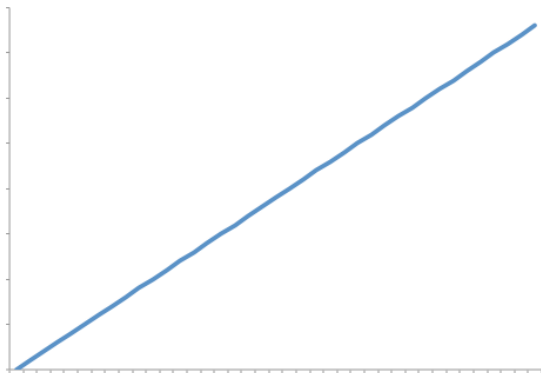
Growth Rates

Constant



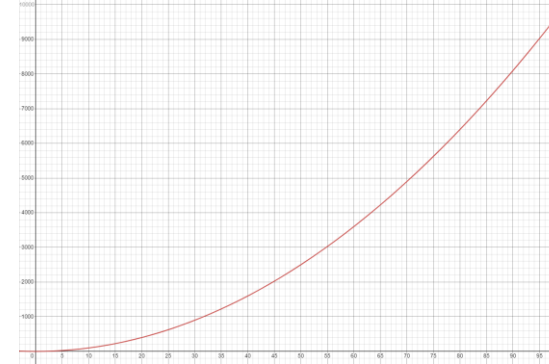
Adding to front of linked list

Linear



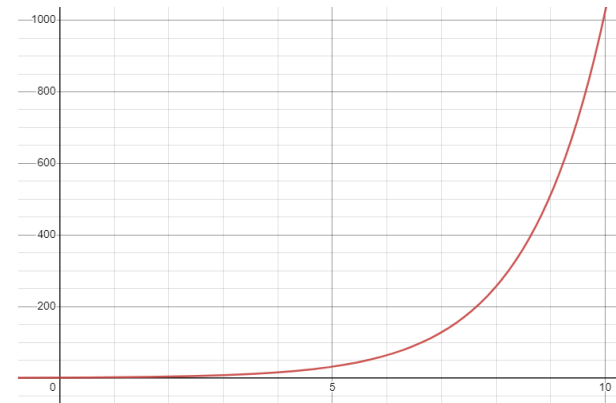
Searching an array for a certain element

Quadratic



Printing out a 2D array

Exponential



Generating all possible
binary strings of length
 N

The **running time** of an algorithm is the time it takes for an algorithm to completely run from start to finish

There are a few ways we can measure running time:

- ~~1. Time (seconds, nanoseconds, minutes, days, etc)~~
2. Number of **operations** required to complete algorithm.

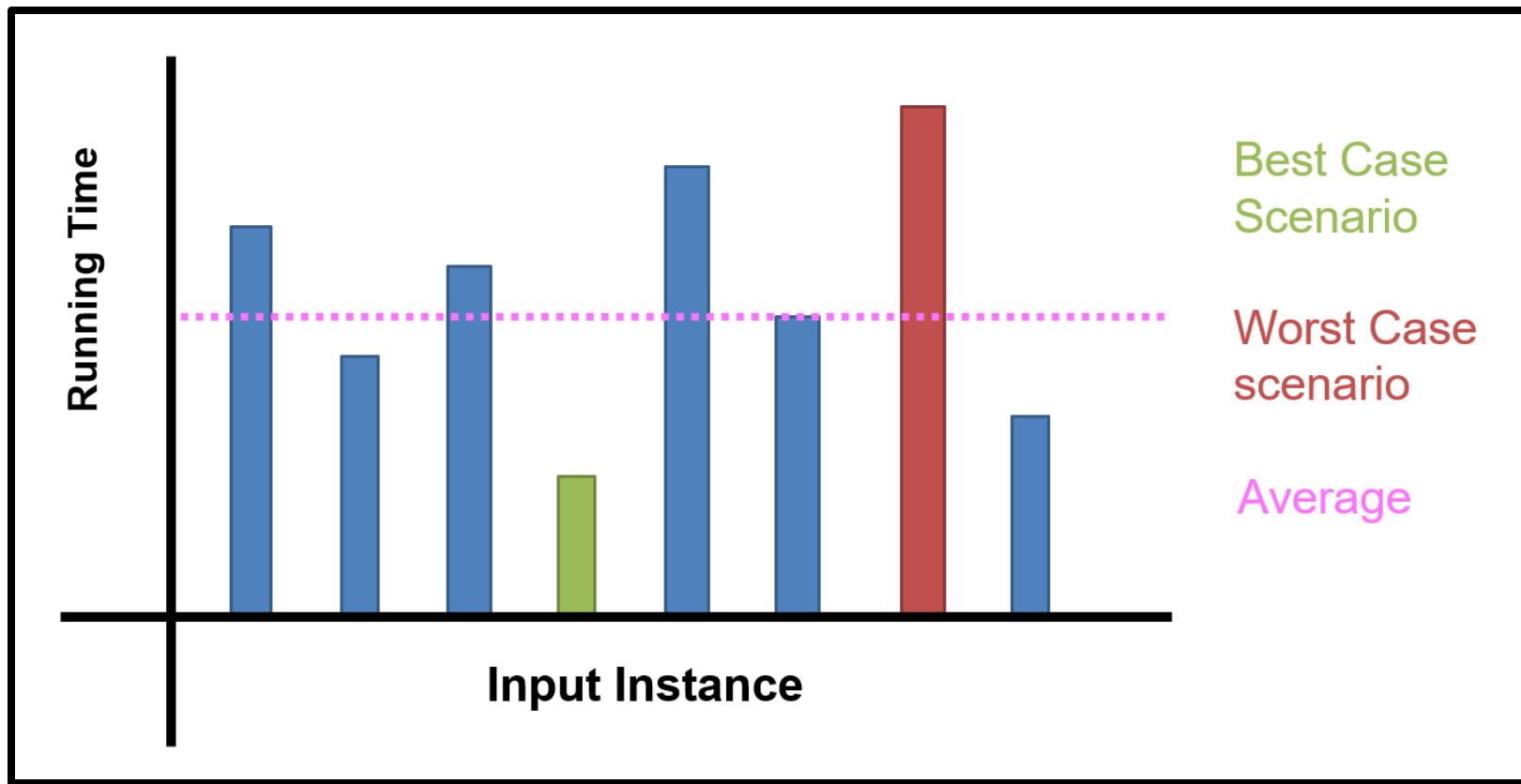
To measure the running time of an algorithm, we will count the number of operations the algorithm performs, and look at how these operations scale *as the input increases*

When we describe the running time of an algorithm, we will represent it using **Big-O Notation**

A **primitive operation** is an operation that has a **constant** execution time

- Assigning a value to a variable
- Performing an arithmetic operation
- Comparing two numbers/values
- Accessing an element in an array (by index)
- Calling a method
- Returning from a method
- Printing out a value

```
int N = 3;  
a = a + 3 * 12  
if(n >= i)  
i = arr[3]  
e.print2Darray(array);  
return  
System.out.println("Hi")
```



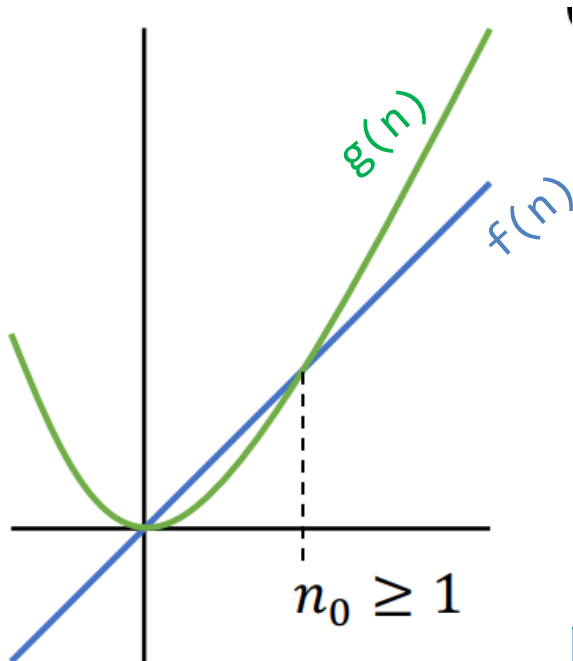
In computer science (and this class in particular), we will be focusing on stating running time in terms of **worst-case scenario**

Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers
 $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Past a certain spot, $g(n)$ dominates $f(n)$ within a multiplicative constant



$$\forall n \geq 1, n^2 \geq n \\ \Rightarrow n \in O(n^2)$$

O -notation provides an upper bound on some function $f(n)$

Big-O

Notation used to describe the running time of an algorithm in terms of worse case scenario

Traits of Big-O-Notation:

In Big-O, we can drop non-dominant factors

In Big-O, we can drop multiplicative constants

Algorithm Analysis: Adding value to an Array/ArrayList

`int[] newArray = new int[myArray.length + 1];` ← $O(n)$

`for(int i = 0; i < myArray.length; i++) {` ← $O(n)$
 `newArray[i] = myArray[i];` ← $O(1)$
`}`

`int new_value = 4;` ← $O(1)$
`newArray[myArray.length] = new_value;` ← $O(1)$
`myArray = newArray;` ← $O(1)$

When we write algorithms,
we should still be *aware of*
these coefficients

$$\begin{aligned}\text{Total Running Time} &= n + n * 1 + 1 + 1 + 1 \\ &= 2n + 3\end{aligned}$$



~~$O(2n)$ where n is the size of the array~~ → $O(n)$ where n is the size of the array

```

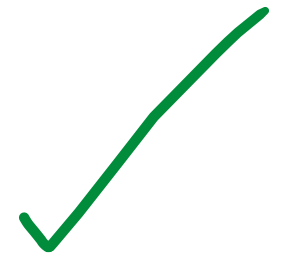
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) { ← N
        if(array[i] == s) {
            return i; } 1
        }
    }
    return -1; ← 1
}

```

Total Running Time = $N + 1$

$O(N + 1)$ where N = Size of Array

$O(N)$ where N = Size of Array



Big-O Complexity Chart

