CS 10:
Problem solving via Object Oriented Programming

Encapsulation

From last class: One way to loop over array elements is to use a C-style for loop

Code

```
public class MultipleVariablesArray {
   public static void main(String[] args) {
      int numberOfScores = 5;
      double[] scores = new double[numberOfScores]; //store quiz scores
      scores[0] = 10;  //zero indexed in Java
      scores[1] = 3.2;
      scores[2] = 6.5;
      scores[3] = 7.8;
      scores[4] = 8.8;  //valid indices are 0..4
      //scores[5] = 9;    //error, index out of bounds!
      System.out.println(scores);

      System.out.print("[");
      for (int i = 0; i < numberOfScores-1; i++) {
         System.out.print(scores[i] + ", ");
      }
      System.out.println(scores[numberOfScores-1] + "]");
   }
}
```

Index	0	1	2	3	4
	10	3.2	6.5	7.8	8.8

Commonly use a variable to declare array size

C-style for loop

Three components:

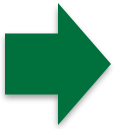
1. **Initialization**
2. **Conditional**
3. **Increment**

Output

```
$ javac MultipleVariablesArray.java
$ java MultipleVariablesArray
D@1dbd16a6
[10.0, 3.2, 6.5, 7.8, 8.8]
```

See Day 1 slides for 2D array, useful for SA-1

Agenda



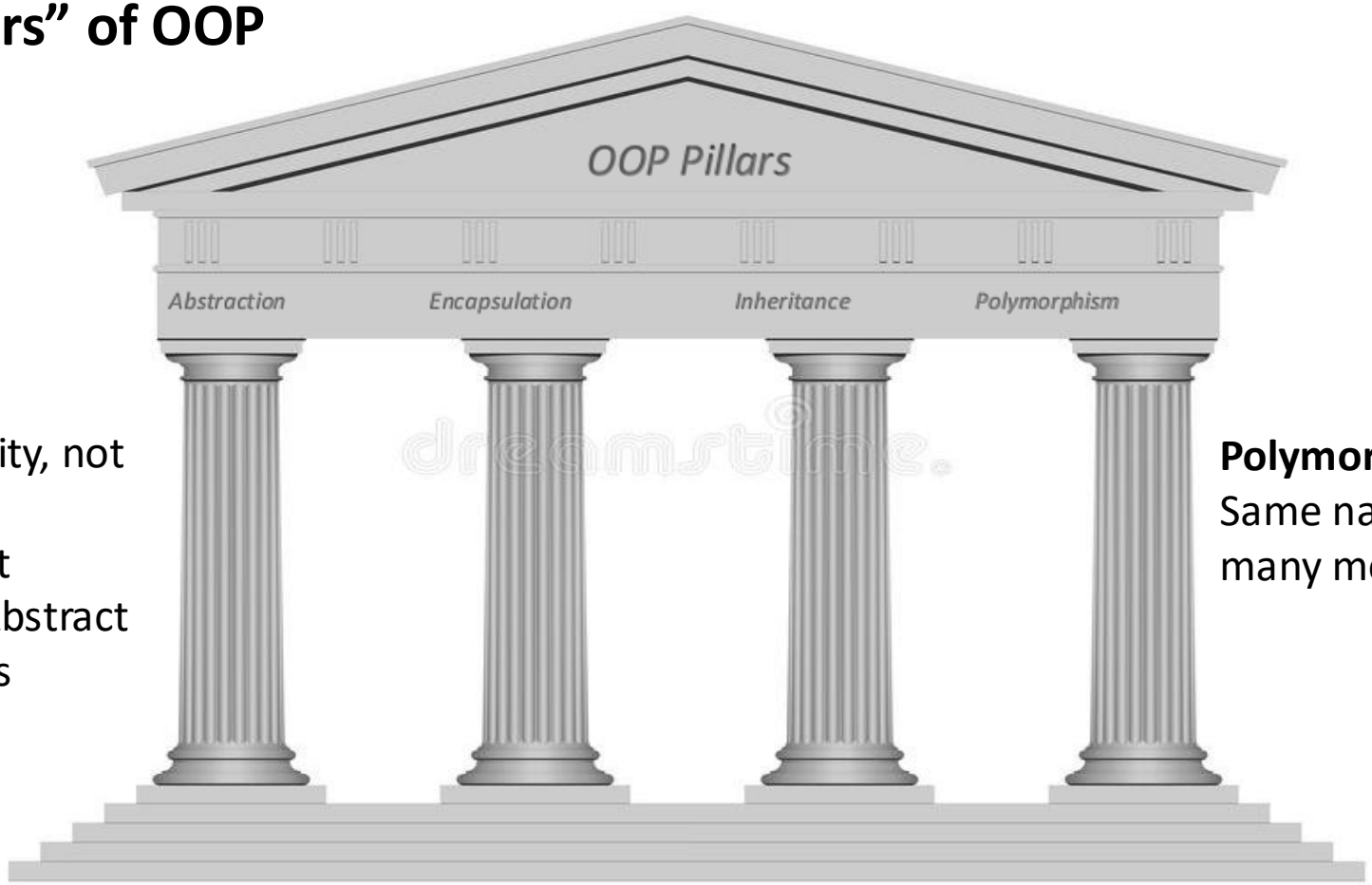
1. Encapsulation
2. Getters/setters
3. Constructors
4. Objects vs. primitives
5. Applications vs classes

Key points:

1. **Encapsulation brings code and data together into one thing called an object**
2. **A class provides a blueprint for instantiating (creating) objects**
3. **An object's data is stored in instance variables**
4. **Many objects can be instantiated from a class, each object gets its own instance variables**

OOP relies on four main pillars to create robust, adaptable, and reusable code

Four "pillars" of OOP



Abstraction

- Name functionality, not how to implement
- Leads to Abstract Data Types (ADTs)

Polymorphism
Same name, many meanings

Encapsulation

- Bind code and data into one thing called an object
- Code called *methods* in OOP (not functions)

Inheritance

- Create specialty versions that "inherit" functionality of parent
- Reduces code

Today we will focus on encapsulation

Encapsulation

- Binds code (methods) and data together into one self-contained “thing”, called an object in Java
- Each object has its own data about itself (e.g., student name and graduation year)
- Objects can make data about itself public or private
- Private data allows an object to control access to data from outside (e.g., if private, then only the object itself can alter its internal data)

We start with a Student class to represent one student

Data
- Name
- Year

Methods (code)
- Study
- Attend class

Example Student class
- We will model students as *objects*
- Objects encapsulate:
  - Data about one student (e.g., name, year)
  - Actions students can take (e.g, study, attendClass) called *methods*
- Objects are defined by a *class*
  - Like a blueprint – a class tells how to create an object (such as a house)
  - A class does not itself create objects
- Each object is *instantiated* (created) from the class in Java using the "new" keyword
- There can be many objects created from the same class (like there can be many houses built from the same blueprint)


ENOUGH TALK
SHOW US HOW IT WORKS
memegenerator.net

Student0.java is our first "real" class represents one student

Student0.java

```
/**
 * Student series demonstrates encapsulation by representing a student in a class
 * Student0 - base example with name and graduation year
 *
 * @author Tim Pierson, Dartmouth CS10, Winter 2025
 */

public class Student0 {
   String name;
   int graduationYear;

   public static void main(String[] args) {
      Student0 alice = new Student0();
      alice.name = "Alice";
      alice.graduationYear = 2027;
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

JavaDoc tells what the program does and who wrote it

- **Starts with /** ends with */**
- **Provides an @author tag describing who wrote it**
- **For problem sets, if you work with a partner, include an @author tag for both partners**

Provide a JavaDoc for the class itself, plus one for each method

Student0.java is our first "real" class represents one student

Student0.java

```
/**
 * Student series demonstrates encapsulation by representing a student in a class
 * Student0 - base example with name and graduation year
 *
 * @author Tim Pierson, Dartmouth CS10, Winter 2025
 */

public class Student0 {
   String name;
   int graduationYear;

   public static void main(String[] args) {
      Student0 alice = new Student0();
      alice.name = "Alice";
      alice.graduationYear = 2027;
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

Class Student0 holds data about one student

Java naming convention:

- **Classes start with a capital letter**
- **Variables use camelCase (not snake_case like Python or C)**
- **I will be looking for you to follow this convention in CS10!**

Each student has “instance variables” that hold data about the student

Student0.java

```
/**
 * Student series demonstrates encapsulation by representing a student in a class
 * Student0 - base example with name and graduation year
 *
 * @author Tim Pierson, Dartmouth CS10, Winter 2025
 */

public class Student0 {
   String name;
   int graduationYear;

   public static void main(String[] args) {
      Student0 alice = new Student0();
      alice.name = "Alice";
      alice.graduationYear = 2027;
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

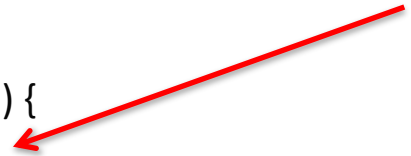
Data is stored in *instance variables*

- **Instance variables are declared outside any method (otherwise they’d be local to the method)**
- **You can think of them like global variables for the class**
- **Track student’s name and graduation year**
- **Must declare data type (String and integer here)**
- **Java initializes instance variables to to 0, null, or false by default (unlike local variables, which are not initialized!)**
- **Each object we create gets its own instance variables**

Student objects are created from the class by using the keyword “new”

Student0.java

```
/**
 * Student series demonstrates encapsulation by representing a student in a class
 * Student0 - base example with name and graduation year
 *
 * @author Tim Pierson, Dartmouth CS10, Winter 2025
 */

public class Student0 {
   String name;
   int graduationYear;

   public static void main(String[] args) {
      Student0 alice = new Student0();
      alice.name = "Alice";
      alice.graduationYear = 2027;
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

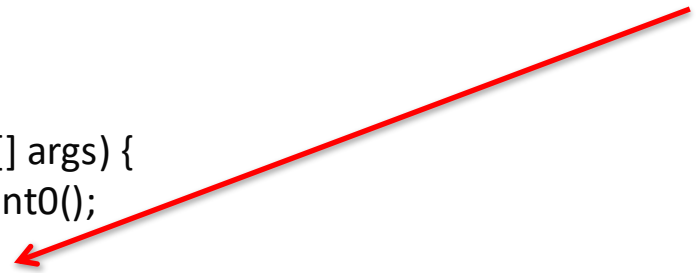


- **“Instance” of class Student0 called “alice” is “instantiated” (created)**
- **Alice’s type is Student0, akin to how *graduationYear* is an integer**
- **Use keyword “new” to create (allocate memory) a new object of type Student0**
- **Java initializes instance variables *name* to *null* and *graduationYear* to 0**

Instance variables *can* be accessed using the dot operator

Student0.java

```
/**
 * Student series demonstrates encapsulation by representing a student in a class
 * Student0 - base example with name and graduation year
 *
 * @author Tim Pierson, Dartmouth CS10, Winter 2025
 */

public class Student0 {
   String name;
   int graduationYear;

   public static void main(String[] args) {
      Student0 alice = new Student0();
      alice.name = "Alice";
      alice.graduationYear = 2027;
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

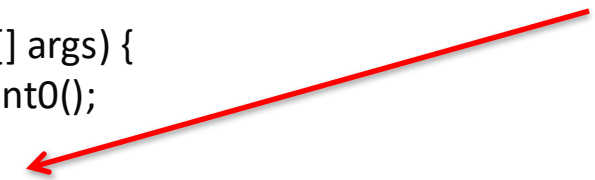


- **Alice's instance variables can be set by using the dot operator**
- **For example: *alice.name***
- **Here Alice's details are then printed to console**

Instance variables *can* be accessed using the dot operator

Student0.java

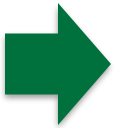
```
/**
 * Student series demonstrates encapsulation by representing a student in a class
 * Student0 - base example with name and graduation year
 *
 * @author Tim Pierson, Dartmouth CS10, Winter 2025
 */

public class Student0 {
   String name;
   int graduationYear;

   public static void main(String[] args) {
      Student0 alice = new Student0();
      alice.name = "Alice";
      alice.graduationYear = 2027;
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```



- **Updating instance variables directly is considered bad form in Java (but not in Python)**
- **We will not do this in Java!**
- **Better to let the objects update own instance variables**
- **We will provide methods (code) that can be called to update instance variables**

Agenda

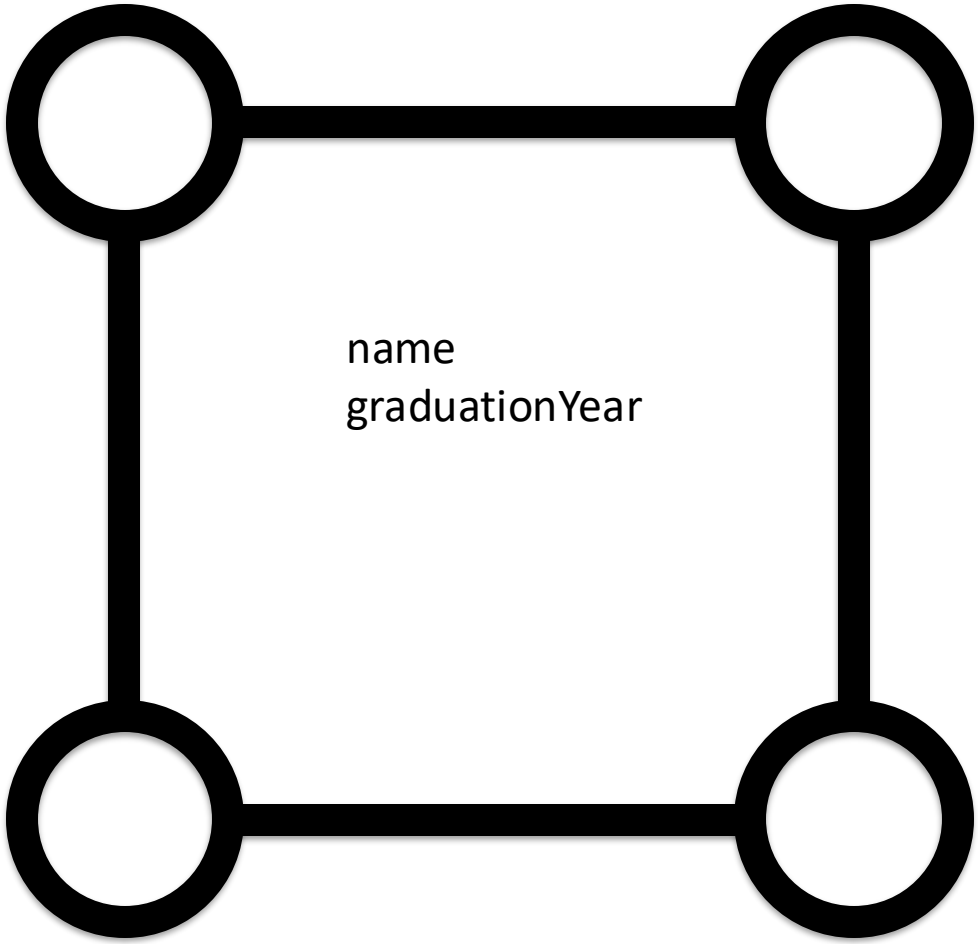
1. Encapsulation
2. Getters/setters
3. Constructors
4. Objects vs. primitives
5. Applications vs classes



Key points:

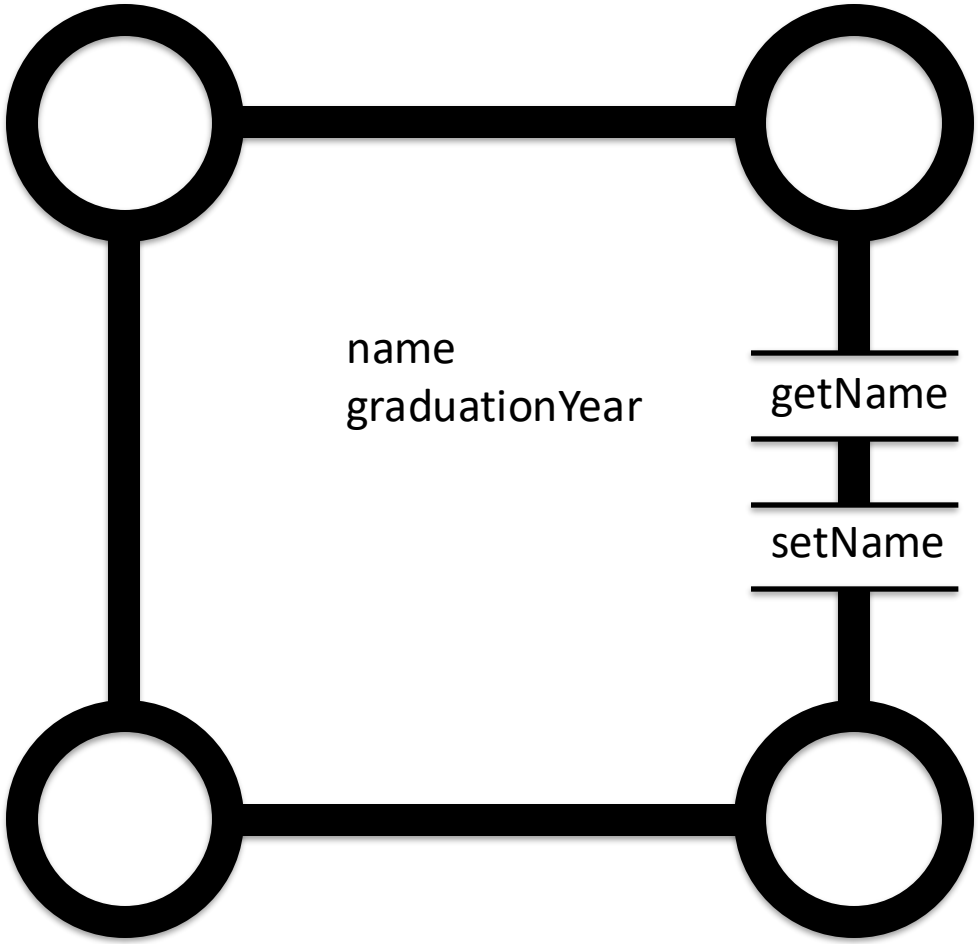
1. **Data can be public or private**
2. **Access to data is normally controlled by getter (return varible's value) and setter (update variable's value) methods**

Pierson's mental model of objects



- **Castle walls protect instance variables from access or modification by outsiders**

Pierson's mental model of objects



- **Castle walls protect instance variables from access or modification by outsiders**
- **Methods (code) allow outsiders to call code to operate on instance variables**
- **Outsiders call *getName* to get the student's name, *setName* to change the student's name**
- **Code can decide if it is going to allow the change**

Access modifiers such as “protected” can prevent outside modification

Student01.java

```
public class Student01 {
   protected String name;
   protected int graduationYear;

   /**
    * Setters for instance variables
    */
   public void setName(String name) { this.name = name; }
   public void setYear(int year) { graduationYear = year; }

   public static void main(String[] args) {
      Student01 alice = new Student01();
      alice.setName("Alice");
      alice.setYear(2027);
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

- ***protected*** **allows this class (and subclasses) to access instance variables**
- **Sets up “castle walls”**
- ***public*** **allows anyone to access instance variables (no walls)**
- **More on this topic soon**

“setter” methods are used to update instance variables instead of dot operator

Student01.java

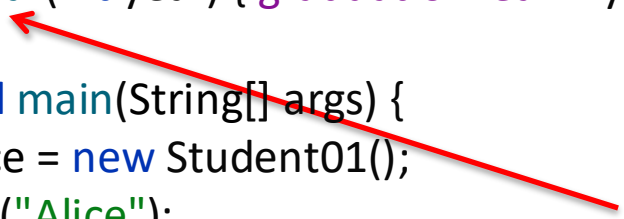
```
public class Student01 {
   protected String name;
   protected int graduationYear;

   /**
    * Setters for instance variables
    */
   public void setName(String name) { this.name = name; }
   public void setYear(int year) { graduationYear = year; }

   public static void main(String[] args) {
      Student01 alice = new Student01();
      alice.setName("Alice");
      alice.setYear(2027);
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

- **“Setter” methods allows object to update its own instance variables based on value passed in as a parameter**
- **Could do error checking here (ex., suppose *year* can’t be negative)**
- **Note the one-line syntax!**
- ***void* means this method does not return a value**

“setter” methods are used to update instance variables instead of dot operator

Student01.java

```
public class Student01 {
   protected String name;
   protected int graduationYear;

   /**
    * Setters for instance variables
    */
   public void setName(String name) { this.name = name; }
   public void setYear(int year) { graduationYear = year; }

   public static void main(String[] args) {
      Student01 alice = new Student01();
      alice.setName("Alice");
      alice.setYear(2027);
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```



- **Setter naming convention: *set<VariableName>***
- **Convention is not enforced by Java**
- ***setYear* would typically be called *setGraduationYear* to match the instance variable name**
- **Java doesn’t care what you call these methods**

Setters usually take a value as a parameter that matches instance variable data type

Student01.java

```
public class Student01 {
   protected String name;
   protected int graduationYear;

   /**
    * Setters for instance variables
    */
   public void setName(String name) { this.name = name; }
   public void setYear(int year) { graduationYear = year; }

   public static void main(String[] args) {
      Student01 alice = new Student01();
      alice.setName("Alice");
      alice.setYear(2027);
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

Instance variable *name* is a String, so the setter parameter is also a String

- **"*this.name*" means the "instance variable *name*" for this object**
- ***name* refers to the parameter**
- **"this" is used when there is ambiguity between variable names (instance variable or parameter here)**
- **"*this*" is like "*self*" in Python**

Setters usually take a value as a parameter that matches instance variable data type

Student01.java

```
public class Student01 {
   protected String name;
   protected int graduationYear;

   /**
    * Setters for instance variables
    */
   public void setName(String name) { this.name = name; }
   public void setYear(int year) { graduationYear = year; }

   public static void main(String[] args) {
      Student01 alice = new Student01();
      alice.setName("Alice");
      alice.setYear(2027);
      System.out.println("Name: " + alice.name +
            ", Year: " + alice.graduationYear);
   }
}
```

- **"this" keyword not needed if there is no ambiguity**
- **Here the parameter's name is *year* and the instance variable is *graduationYear***
- **Java can determine which variable to use based on the name**
- **Above the parameter and instance variable have the same name**

Setters allow an object to decide whether to accept a new value

Student02.java

```
public class Student02 {
   protected String name;
   protected int graduationYear;

    /**
     * Setters for instance variables
     */
   public void setName(String name) { this.name = name; }
   public void setYear(int year) {
      //only accept valid years
      if (year > 1769 && year < 2100) {
         graduationYear = year;
      }
   }

<snip>
```

- ***setYear*** **checks that the year parameter is reasonable**
- **For example, a year of -1 would not make sense**
- **Later we will throw an exception to tell the caller we did not take the new value**
- **For now, we simply ignore invalid values and leave *graduationYear* unchanged for invalid input**

“Getters” return values

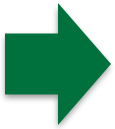
Student03.java

```
public class Student03 {
   protected String name;
   protected int graduationYear;

<snip>

   /**
    * Getters for instance variables
    */
   public String getName() { return name; }
   public int getGraduationYear() { return graduationYear; }

public static void main(String[] args) {
      Student03 alice = new Student03();
      alice.setName("Alice");
      alice.setYear(2027);
```

- **Getters are used to return protected instance variables so other code can see the variable’s value**
- **Naming convention is like setters, *get<VariableName>***
- **Method declarations give the return data type**
- **Java returns one value (unlike Python, which can return multiple values)**
- **Use *void* as a return data type if a method does not return a value**

Agenda

1. Encapsulation
2. Getters/setters
3. Constructors
4. Objects vs. primitives
5. Applications vs classes



Key points:

1. **Constructors are a way to initialize new objects**
2. **Constructors are called when an object is first instantiated**

Constructors allow us to initialize an object when it is instantiated

Student03.java

```
public class Student03 {
   protected String name;
   protected int graduationYear;

<snip>

   /**
    * Getters for instance variables
    */
   public String getName() { return name; }
   public int getGraduationYear() { return graduationYear; }

public static void main(String[] args) {
      Student03 alice = new Student03();
      alice.setName("Alice");
      alice.setYear(2027);
```

- **Remember: by default, instance variables are initialized to 0, null, or false**
- **So, *name* is *null* and *graduationYear* is 0 here**
- **This is not what we normally want**
- **It would be tedious to call the setter for each instance variable**
- **We have a better way – constructors!**

Constructors run first when an object it is instantiated; allow object initialization

Student04.java

```
public class Student04 {
   protected String name;
   protected int graduationYear;

   public Student04() {
      //default constructor: you get this by default
   }

   public Student04(String name, int year) {
      this.name = name;
      graduationYear = year;
   }

<snip>

public static void main(String[] args) {
   Student04 abby = new Student04();
   Student04 alice = new Student04("Alice", 2027);
```

- **Constructors have same name as class**
- **Run when object is first instantiated**
- **This constructor takes no parameters**
- ***name* is null, *graduationYear* is 0**
- **If you don't provide any constructors, then you *implicitly* get one like this**

Overload constructors (and other methods) to create multiple method versions

Student04.java

```
public class Student04 {
   protected String name;
   protected int graduationYear;

   public Student04() {
      //default constructor: you get this by default
   }

   public Student04(String name, int year) {
      this.name = name;
      graduationYear = year;
   }

<snip>

public static void main(String[] args) {
   Student04 abby = new Student04();
   Student04 alice = new Student04("Alice", 2027);
```

- **This constructor takes two parameters, one for each instance variable**
- **Multiple methods with same name is called *overloading***
- **Java determines which to use based on parameters provided when called (signature)**

Overload constructors (and other methods) to create multiple method versions

Student04.java

```
public class Student04 {
   protected String name;
   protected int graduationYear;

   public Student04() {
      //default constructor: you get this by default
   }

   public Student04(String name, int year) {
      this.name = name;
      graduationYear = year;
   }

<snip>

public static void main(String[] args) {
   Student04 abby = new Student04();
   Student04 alice = new Student04("Alice", 2027);
```

- **When *abby* is instantiated, Java calls the first constructor (no parameters provided)**
- **When *alice* is instantiated, Java calls the second constructor (String and an int)**
- **What values do *abby's* instance variables hold? *alice's*?**

Objects can have other methods other than a constructor

Student05.java

```
public class Student05 {
   protected String name;
   protected int graduationYear;
   protected double studyHours;
   protected double classHours;

<snip>

   /**
    * Getters for instance variables
    */
   public String getName() { return name; }
   public int getGraduationYear() { return graduationYear; }
   public double getstudyHours() { return studyHours; }
   public double getclassHours() { return classHours; }
```

Add new instance variables to track hours studying and in class

Add getters for new instance variables

Objects can have other methods other than a constructor

Student05.java

```
 /**
 * adds hoursSpent to the studyHours to track time this student spent studying
 * @param hoursSpent - number of hours spent studying (can have decimal component)
 * @return - total number of hours spent studying including the new hours passed in
 */
public double study(double hoursSpent) {
   System.out.println("Hi Mom! It's " + name +". I'm in studying!");
   studyHours += hoursSpent;
   return studyHours;
}

/**
 * adds hoursSpent to the classHours to track time this student spent in class
 * @param hoursSpent - number of hours spent in class (can have decimal component)
 * @return - total number of hours spent in class including the new hours passed in
 */
public double attendClass(double hoursSpent) {
   System.out.println("Hi Dad! It's " + name +". I'm in class!");
   classHours += hoursSpent;
   return classHours;
}
```

Don't forget JavaDocs

***study* method**

- **Alerts Mom**
- **adds hours spent studying**
- **Returns total hours spent studying**

***attendClass* method**
Like study method but alerts Dad

Agenda

1. Encapsulation
2. Getters/setters
3. Constructors
4. Objects vs. primitives
5. Applications vs classes

Key points:

1. **Primitive variables are stored on the stack**
2. **Objects are stored in the heap**

Recall: Java defines several primitive types, each of fixed memory size

Common primitive types

Type	Description	Size	Examples
int	Integer values (no decimal component)	32 bits (4 bytes)	-104,…1,2,3…107,…5032
double	Double precision floating point (has decimal component)	64 bits (8 bytes)	-123.45, 1.6
boolean	true or false	1 bit	true, false
char	Characters	16 bits (2 bytes for Unicode)	‘a’,’b’,…’z’

NOTE: Java provides an Object wrapper for each primitive (called autoboxing). Reference them with an initial capital letter (e.g., Integer, Double, Boolean, Character)

Declaring a primitive variable allocates stack space that holds variable's value

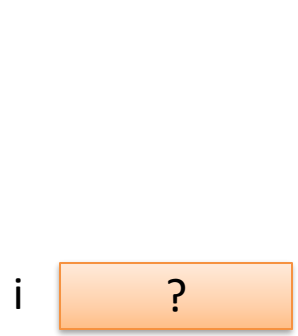
```
public class MemoryAllocationPrimitives {

   public static void main(String[] args) {
     //declare local variables
     int i; double d; boolean b; char c;

     //assign values to local variables
     i=7; d=1.6; b=true; c='a';

     //print new values
     System.out.println("Local variables: "+
          "i="+i+" d="+d+" b="+b+" c="+c);
   }
}
```

Stack **Heap**

- **Stack is where Java keeps track of its variables**
- **Heap is for dynamic memory allocation (take CS50 for more)**

Declaring a primitive variable allocates stack space that holds variable's value

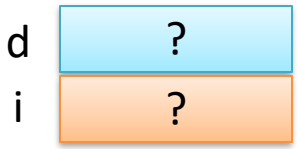
```
public class MemoryAllocationPrimitives {

   public static void main(String[] args) {
     //declare local variables
     int i; double d; boolean b; char c;

     //assign values to local variables
     i=7; d=1.6; b=true; c='a';

     //print new values
     System.out.println("Local variables: "+
         "i="+i+" d="+d+" b="+b+" c="+c);
   }
}
```



- **While executing line, space is allocated on the stack for the primitive local variables**
- **Java doesn't initialize local variables (like it does instance variables)**
- **Exception (error) raised if try to use a local variable before value assigned (e.g., i=i+1; is exception)**

Declaring a primitive variable allocates stack space that holds variable's value

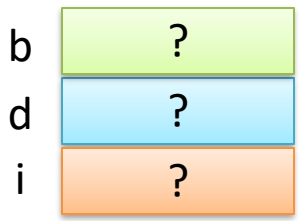
```
public class MemoryAllocationPrimitives {

   public static void main(String[] args) {
     //declare local variables
     int i; double d; boolean b; char c;

     //assign values to local variables
     i=7; d=1.6; b=true; c='a';

     //print new values
     System.out.println("Local variables: "+
          "i="+i+" d="+d+" b="+b+" c="+c);
   }
}
```



- **While executing line, space is allocated on the stack for the primitive local variables**
- **Java doesn't initialize local variables (like it does instance variables)**
- **Exception (error) raised if try to use a local variable before value assigned (e.g., i=i+1; is exception)**
- **NOTE: showing primitive types as same size for convenience**

Declaring a primitive variable allocates stack space that holds variable's value

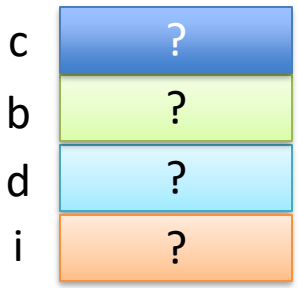
```
public class MemoryAllocationPrimitives {

   public static void main(String[] args) {
     //declare local variables
     int i; double d; boolean b; char c;

     //assign values to local variables
     i=7; d=1.6; b=true; c='a';

     //print new values
     System.out.println("Local variables: "+
          "i="+i+" d="+d+" b="+b+" c="+c);
   }
}
```



- **While executing line, space is allocated on the stack for the primitive local variables**
- **Java doesn't initialize local variables (like it does instance variables)**
- **Exception (error) raised if try to use a local variable before value assigned (e.g., i=i+1; is exception)**
- **NOTE: showing primitive types as same size for convenience**

Declaring a primitive variable allocates stack space that holds variable's value

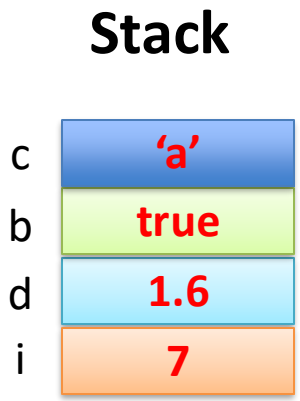
```
public class MemoryAllocationPrimitives {

   public static void main(String[] args) {
     //declare local variables
     int i; double d; boolean b; char c;

     //assign values to local variables
     i=7; d=1.6; b=true; c='a';

     //print new values
     System.out.println("Local variables: "+
          "i="+i+" d="+d+" b="+b+" c="+c);
   }
}
```



- **While executing line, space is allocated on the stack for the primitive local variables**
- **Java doesn't initialize local variables (like it does instance variables)**
- **Exception (error) raised if try to use a local variable before value assigned (e.g., i=i+1; is exception)**
- **NOTE: showing primitive types as same size for convenience**

Declaring a primitive variable allocates stack space that holds variable's value

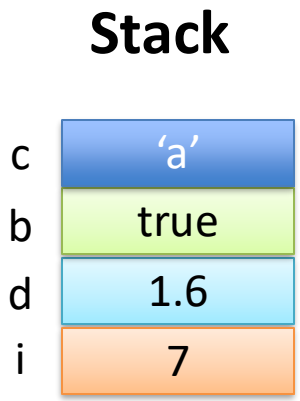
```
public class MemoryAllocationPrimitives {

   public static void main(String[] args) {
     //declare local variables
     int i; double d; boolean b; char c;

     //assign values to local variables
     i=7; d=1.6; b=true; c='a';

     //print new values
     System.out.println("Local variables: "+
          "i="+i+" d="+d+" b="+b+" c="+c);
   }
}
```



- **After executing line, values assigned to primitive local variables**

Declaring a primitive variable allocates stack space that holds variable's value

```
public class MemoryAllocationPrimitives {

   public static void main(String[] args) {
     //declare local variables
     int i; double d; boolean b; char c;

     //assign values to local variables
     i=7; d=1.6; b=true; c='a';

     //print new values
     System.out.println("Local variables: "+
         "i="+i+" d="+d+" b="+b+" c="+c);
   }
}
```



- **Stack holds the *values* of the primitive data types**
- **Printing a primitive type prints its value**

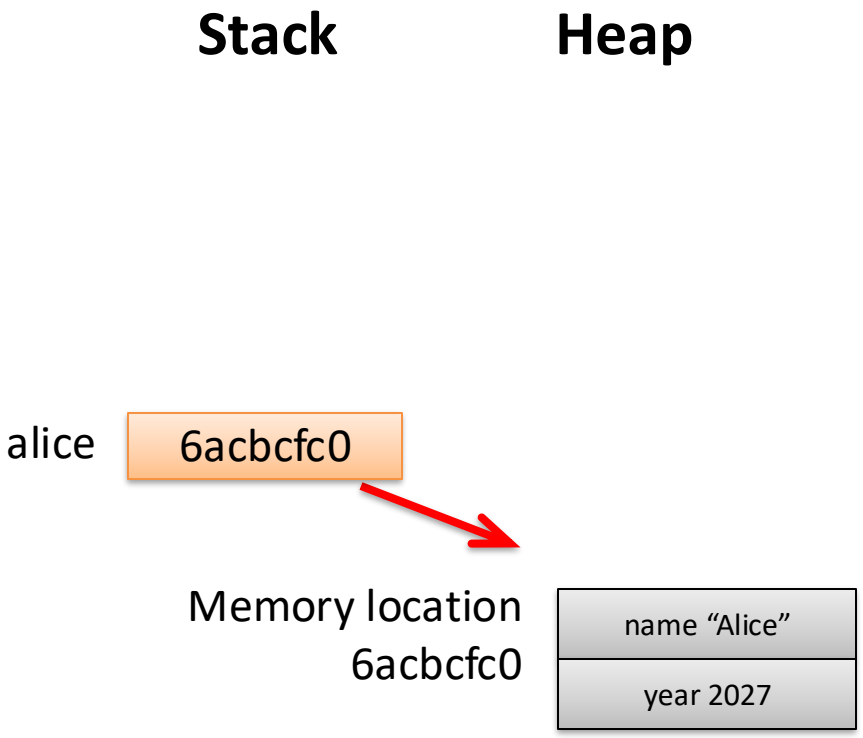
Output

Local variables: i=7 d=1.6 b=true c=a

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {
   //declare Student objects
   Student05 alice = new Student05("Alice", 2027);
```

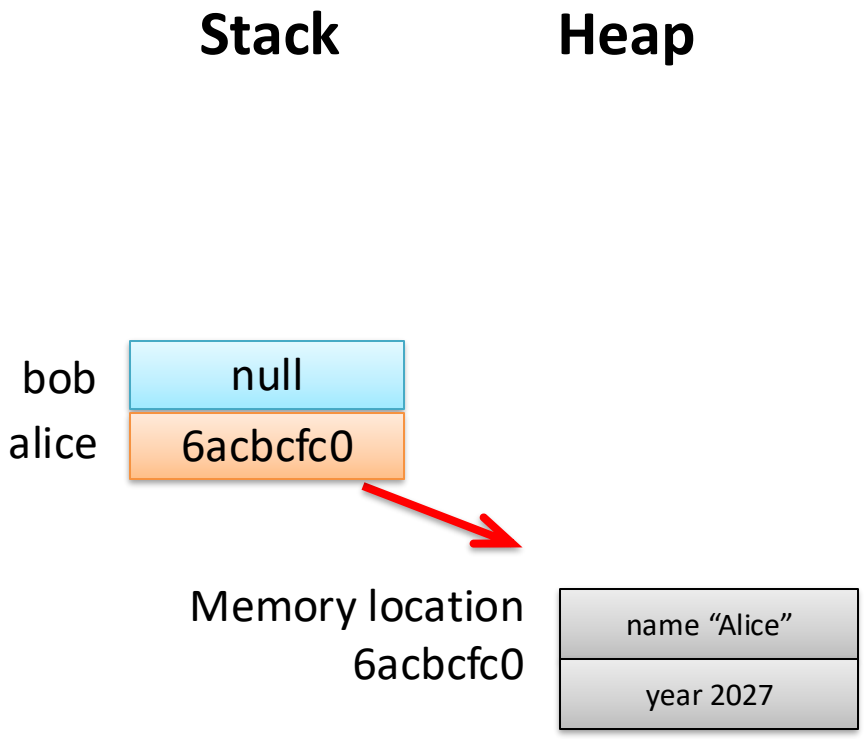
- **Stack holds *memory address* of object (with primitives, stack holds variable's *value*)**
- **Memory address tells Java where to find the "alice" object in memory**
- **Object itself allocated elsewhere in memory (in heap, not on stack)**
- **OS chooses where to allocate**



Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {
    //declare Student objects
    Student05 alice = new Student05("Alice", 2027);
    Student05 bob; //notice no new keyword
```

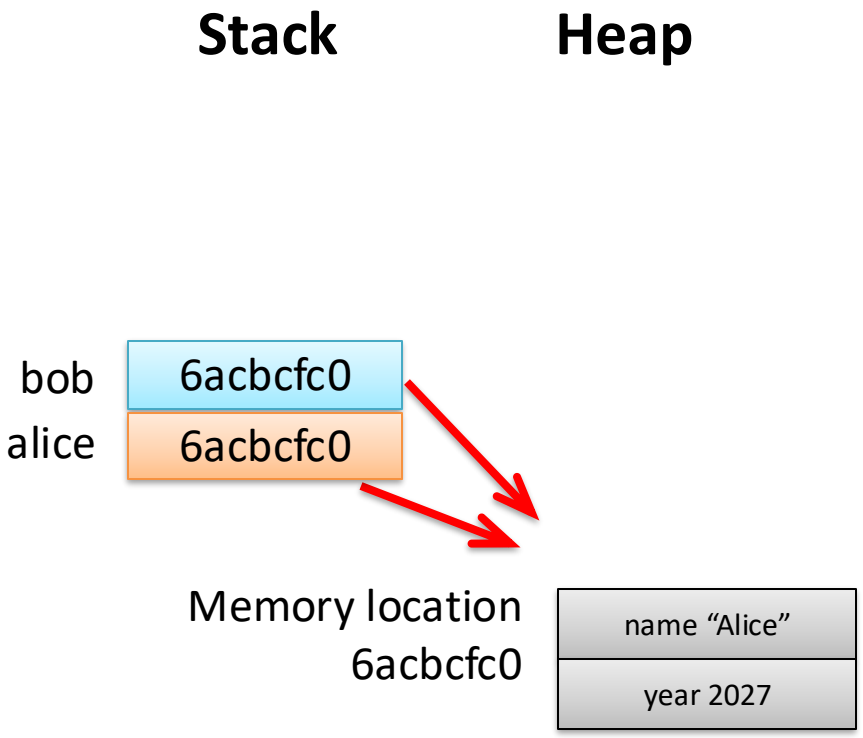
- **"bob" is allocated on the stack, but is null (points nowhere)**
- **Null means "no value"**
- **This is because *bob* did not use the "new" keyword to allocate memory and cause the constructor to run**
- **Null pointer exception if try to use *bob* now**



Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {
    //declare Student objects
    Student05 alice = new Student05("Alice", 2027);
    Student05 bob; //notice no new keyword
    bob = alice; //bob equals alice
```

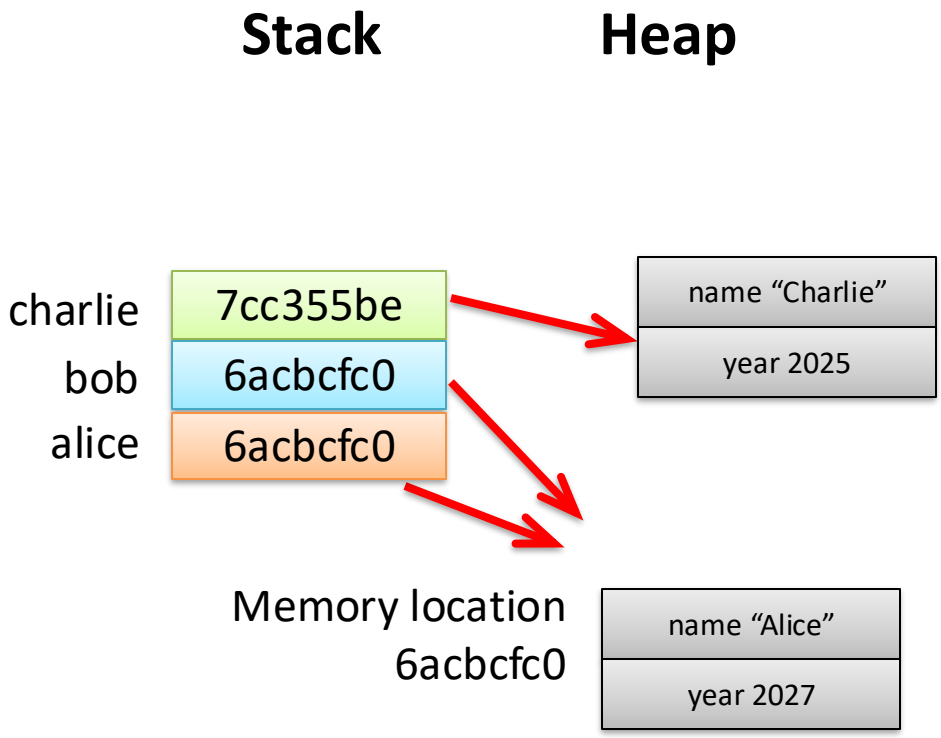
- ***bob* set equal to *alice***
- ***bob* gets same value on stack that *alice* holds**
- ***bob* now references to the exact same memory location as *alice***
- ***bob* and *alice* are "aliases" of each other**



Declaring objects makes pointer on the stack, but object itself is elsewhere

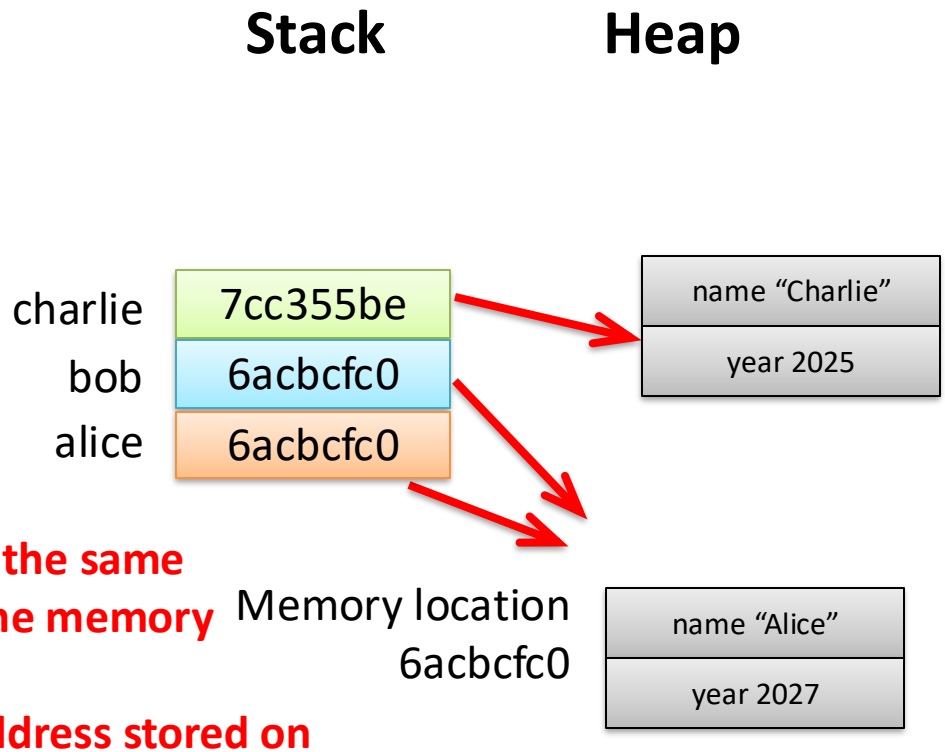
```
public static void main(String[] args) {
    //declare Student objects
    Student05 alice = new Student05("Alice", 2027);
    Student05 bob; //notice no new keyword
    bob = alice; //bob equals alice
    Student05 charlie = new Student05("Charlie", 2025);
```

***charlie* object gets new allocation elsewhere in memory because "new" keyword used**



Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {
    //declare Student objects
    Student05 alice = new Student05("Alice", 2027);
    Student05 bob; //notice no new keyword
    bob = alice; //bob equals alice
    Student05 charlie = new Student05("Charlie", 2025);
    System.out.println(alice.name +" "+bob.name);
```

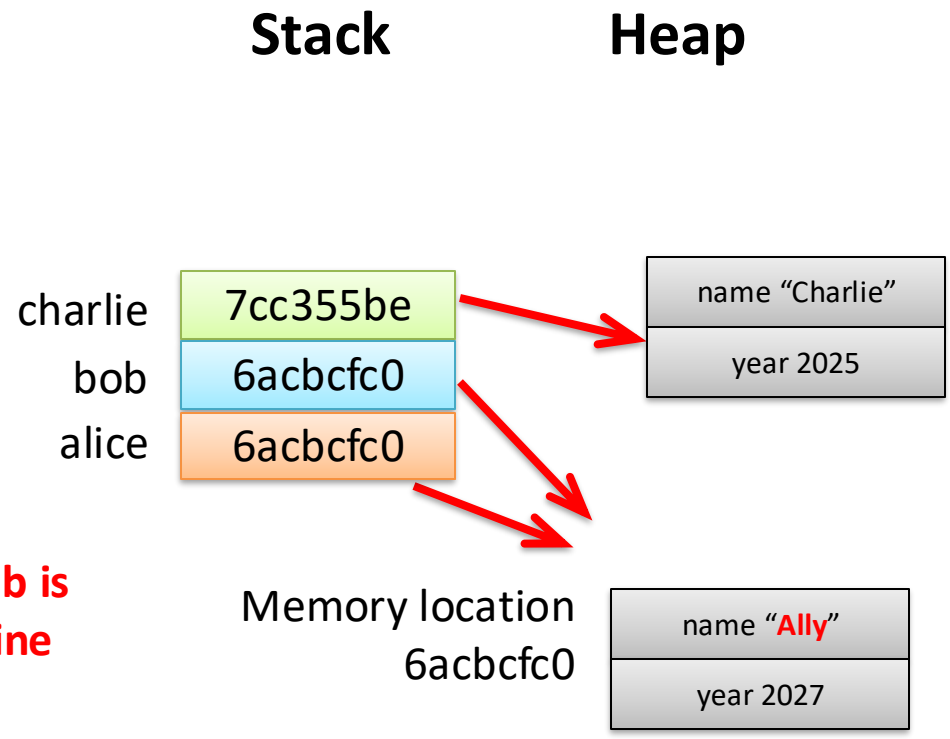


Output
Alice Alice

- ***name* value for *alice* and *bob* is the same because stored at the exact same memory address**
- **Java “dereferences” memory address stored on the stack to find objects on the heap**
- **For *alice* Java goes to memory location 6acbcfc0 on the heap and prints the name stored there**
- **Does the same for *bob***
- **Java doesn’t know or care that *alice* and *bob* both reference the same memory on the heap**

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {
    //declare Student objects
    Student05 alice = new Student05("Alice", 2027);
    Student05 bob; //notice no new keyword
    bob = alice; //bob equals alice
    Student05 charlie = new Student05("Charlie", 2025);
    System.out.println(alice.name +" "+bob.name);


    //update alice's name
    alice.setName("Ally");
```

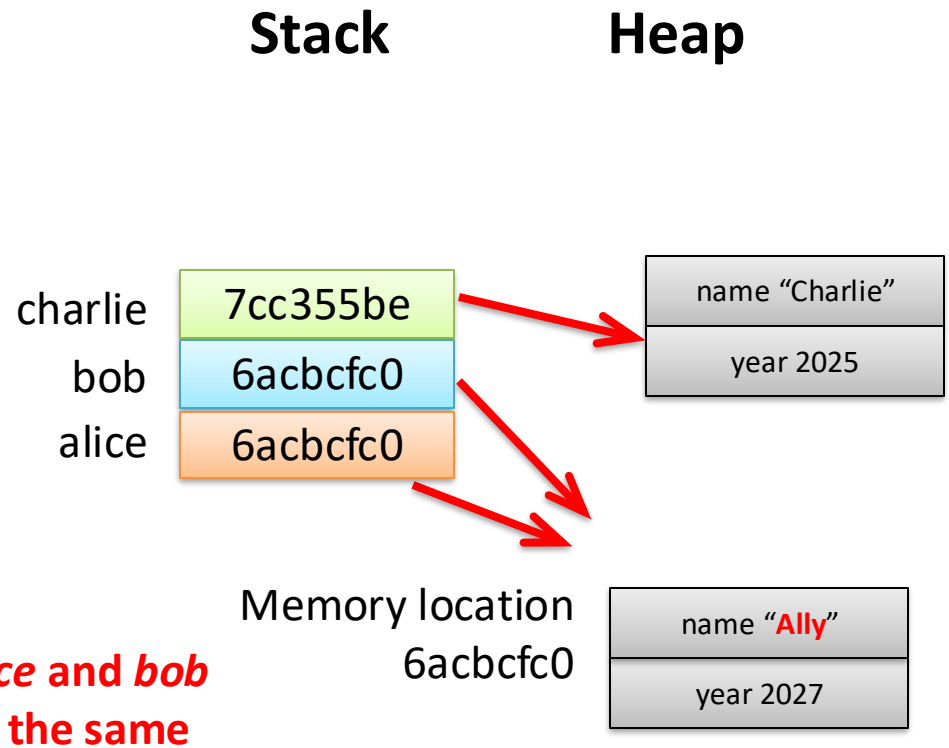


Output
Alice Alice

- ***alice.name* set to "Ally" but Bob is not explicitly changed by this line of code**
- **Expect *alice's* name to change**
- **What about *bob*?**

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {
    //declare Student objects
    Student05 alice = new Student05("Alice", 2027);
    Student05 bob; //notice no new keyword
    bob = alice; //bob equals alice
    Student05 charlie = new Student05("Charlie", 2025);
    System.out.println(alice.name +" "+bob.name);


    //update alice's name
    alice.setName("Ally");
    System.out.println(alice.name +" "+bob.name);
```



Output

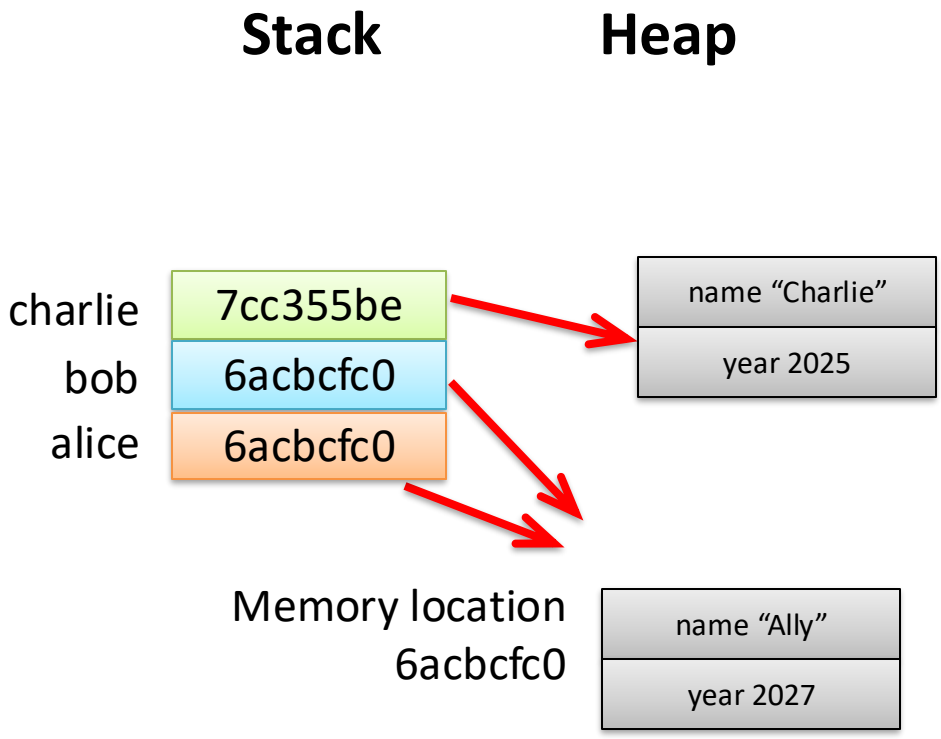
Alice Alice

Ally Ally

- ***name* is the same for both *alice* and *bob* objects because they point to the same memory address (aliases)**
- **Changing one changes the other**
- **Like Python setting two lists equal to each other, change one list, change the other also**
- ***charlie's* name is still "Charlie"**

Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {
    //declare Student objects
    Student05 alice = new Student05("Alice", 2027);
    Student05 bob; //notice no new keyword
    bob = alice; //bob equals alice
    Student05 charlie = new Student05("Charlie", 2025);
    System.out.println(alice.name +" "+bob.name);

    //update alice's name
    alice.setName("Ally");
    System.out.println(alice.name+" "+bob.name);

    //printing objects implicitly calls toString()
    System.out.println(alice+" "+bob+" "+charlie);
}
```



Output

Alice Alice

Ally Ally

Student05@6acbcfc0 Student05@6acbcfc0 Student05@7cc355be

- **Printing an object causes an implicit call to “*toString()*” function**
- **This can be overriden (see course webpage)**
- **By default *toString()* prints memory address of object (for primitives, value is printed)**

Special *toString* method returns a String representation of an object

Student06.java

```
/**
 * Return a String representation of a student
 * @return - string representing the student
 */
public String toString() {
   String s = "Name: " + name + ", graduation year: "
              + graduationYear + "\n";
   s += "\tHours studying: " + studyHours + "\n";
   s += "\tHours in class: " + classHours;
   return s;
}

public static void main(String[] args) {
   Student06 alice = new Student06("Alice", 2027); //calls first constructor
   alice.study(1.5);
   alice.attendClass(1.1);
   System.out.println(alice);
}
```

- ***toString*** **is called when an object is printed**
- **Method should return a string representation of the object**

If toString not provided, Java prints value based on memory address

"\n" is a new line character
"\t" is a tab character

Return String

Output

```
Name: Alice, graduation year: 2027
        Hours studying: 1.5
        Hours in class: 1.1
```

New line character puts following text on next line

Tab character indents

DO NOT print in toString method!

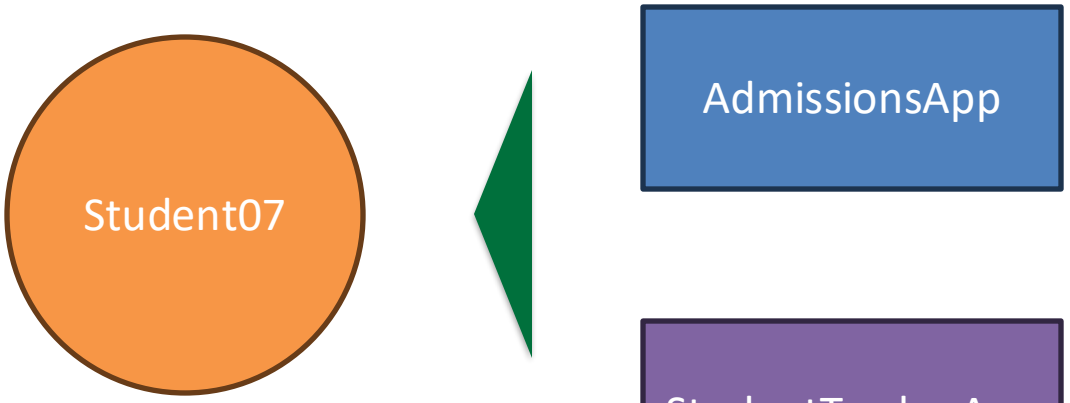
Agenda

1. Encapsulation
2. Getters/setters
3. Constructors
4. Objects vs. primitives
5. Applications vs classes

Key points:

1. **Multiple applications can use the same class**

Frequently we will create classes and use them in application or driver programs



Student07 **class does not have a *main* method**

It is used to model one student at a college

Application programs can then use the class to create objects to suit their business logic

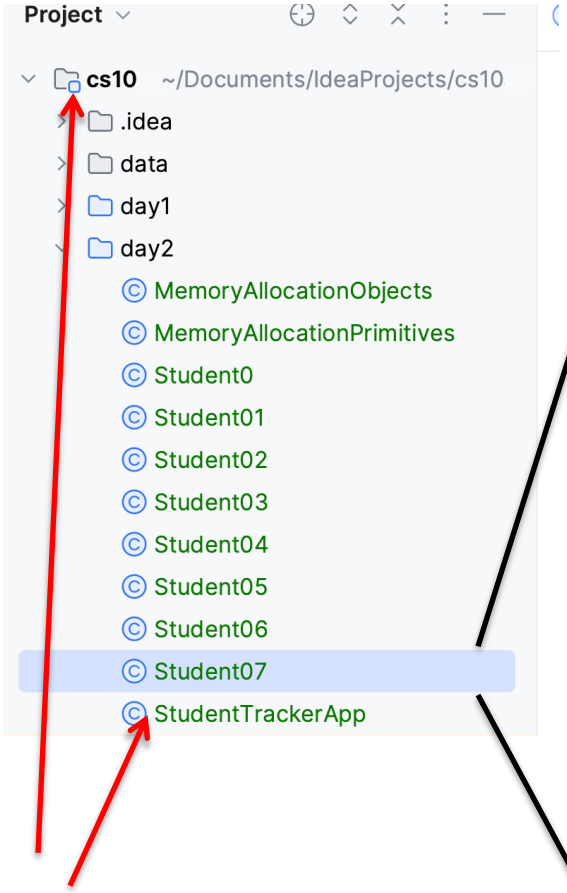
Application programs (sometimes called "driver" programs) provide the business logic to accomplish a task

Here we have an application for a college's Admissions office

Another application might be used by the Registrar to track students

Multiple application programs can use the same Student class

Frequently we will create classes and use them in application or driver programs



***StudentTrackerApp* can use *Student07* class because they are both in the *cs10* project (no need to import)**

```
public class Student07 {
   protected String name;
   protected int graduationYear;
   double studyHours;
   double classHours;

   public Student07() {
      //default constructor: you get this by default
   }

   public Student07(String name, int year) {
      this.name = name;
      graduationYear = year;
   }

<snip>

   public double study(double hoursSpent) {
      System.out.println("Hi Mom! It's " + name + ". I'm studying!");
      studyHours += hoursSpent;
      return studyHours;
   }

   public double attendClass(double hoursSpent) {
      System.out.println("Hi Dad! It's " + name +". I'm in class!");
      classHours += hoursSpent;
      return classHours;
   }

   public String toString() {
      String s = "Name: " + name + ", graduation year: " + graduationYear + "\n";
      s += "\tHours studying: " + studyHours + "\n";
      s += "\tHours in class: " + classHours;
      return s;
   }
}
```

- ***Student07* class does not have a *main* method**
- **Note: JavaDocs and getters/setters removed to fit on the slide!**

- **Java allows classes to use another class if they are both in the same project (cs10 if you followed my instructions to set up IntelliJ)**

StudentTrackerApp is an application that uses the Student07 class

StudentTrackerApp.java

```
public class StudentTrackerApp {
   public static void main(String[] args) {
      int numberOfStudents = 3;
      Student07[] students = new Student07[numberOfStudents];
      students[0] = new Student07("Alice", 2027);
      students[1] = new Student07("Bob", 2024);
      students[2] = new Student07("Charlie", 2025);

      //print students with for-each loop
      System.out.println("Before studying");
      for (Student07 student : students) {
         System.out.println(student);
      }
```

No need to import *Student* class, Java finds it if in the same project

Create an array of Student07 objects

- **Print each student before studying**
- **Here we use a "for-each" loop**
  - **Get first student at index 0**
  - **Store in local variable called *student* of type *Student07***
  - **Print *student* object calls *toString* method**
  - **Get next student until all students processed**
  - **Variable named *student* goes out of scope when loop ends**

StudentTrackerApp is an application that uses the Student07 class

StudentTrackerApp.java

```
public class StudentTrackerApp {
   public static void main(String[] args) {
      int numberOfStudents = 3;
      Student07[] students = new Student07[numberOfStudents];
      students[0] = new Student07("Alice", 2027);
      students[1] = new Student07("Bob", 2024);
      students[2] = new Student07("Charlie", 2025);

      //print students with for-each loop
      System.out.println("Before studying");
      for (Student07 student : students) {
         System.out.println(student);
      }
```

Output

```
Before studying
Name: Alice, graduation year: 2027
        Hours studying: 0.0
        Hours in class: 0.0
Name: Bob, graduation year: 2024
        Hours studying: 0.0
        Hours in class: 0.0
Name: Charlie, graduation year: 2025
        Hours studying: 0.0
        Hours in class: 0.0
```

StudentTrackerApp is an application that uses the Student07 class

```
public class StudentTrackerApp {
   public static void main(String[] args) {
      int numberOfStudents = 3;
      Student07[] students = new Student07[numberOfStudents];
      students[0] = new Student07("Alice", 2027);
      students[1] = new Student07("Bob", 2024);
      students[2] = new Student07("Charlie", 2025);

      //print students with for-each loop
      System.out.println("Before studying");
      for (Student07 student : students) {
         System.out.println(student);
      }

      //randomly select students to study to simulate an actual application
      for (int i = 0; i < 10; i++) {
         //pick random student
         int index = (int)(Math.random() * numberOfStudents);
         Student07 student = students[index];

         //add random studying time between 0 and 5 hours
         double time = Math.random() * 5;
         student.study(time);
      }
```

StudentTrackerApp.java

Output

```
Before studying
Name: Alice, graduation year: 2027
        Hours studying: 0.0
        Hours in class: 0.0
Name: Bob, graduation year: 2024
        Hours studying: 0.0
        Hours in class: 0.0
Name: Charlie, graduation year: 2025
        Hours studying: 0.0
        Hours in class: 0.0
```

Loop 10 times to simulate a real application doing some work

- **Randomly pick one student to study**
- ***Math.random* returns double [0…1)**
- **Multiply by *numberOfStudents***
- **Notice last index at 2, but multiply random by 3 to get value [0…3)**
- **Cast double to integer for index (truncate decimal component)**
- **Get student at randomly chosen index**

- **Simulate student studying for random amount of time between 0 and 5 hours**
- **Call study method on Student to track hours spent studying**

StudentTrackerApp is an application that uses the Student07 class

StudentTrackerApp.java

```
public class StudentTrackerApp {
   public static void main(String[] args) {
      int numberOfStudents = 3;
      Student07[] students = new Student07[numberOfStudents];
      students[0] = new Student07("Alice", 2027);
      students[1] = new Student07("Bob", 2024);
      students[2] = new Student07("Charlie", 2025);

      //print students with for-each loop
      System.out.println("Before studying");
      for (Student07 student : students) {
         System.out.println(student);
      }

      //randomly select students to study to simulate an actual application
      for (int i = 0; i < 10; i++) {
         //pick random student
         int index = (int)(Math.random() * numberOfStudents);
         Student07 student = students[index];

         //add random studying time between 0 and 5 hours
         double time = Math.random() * 5;
         student.study(time);
      }
```

Output

```
Before studying
Name: Alice, graduation year: 2027
        Hours studying: 0.0
        Hours in class: 0.0
Name: Bob, graduation year: 2024
        Hours studying: 0.0
        Hours in class: 0.0
Name: Charlie, graduation year: 2025
        Hours studying: 0.0
        Hours in class: 0.0
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Charlie. I'm studying!
Hi Mom! It's Charlie. I'm studying!
Hi Mom! It's Charlie. I'm studying!
```

StudentTrackerApp is an application that uses the Student07 class

StudentTrackerApp.java

```
public class StudentTrackerApp {
   public static void main(String[] args) {
      int numberOfStudents = 3;
      Student07[] students = new Student07[numberOfStudents];
      students[0] = new Student07("Alice", 2027);
      students[1] = new Student07("Bob", 2024);
      students[2] = new Student07("Charlie", 2025);

      //print students with for-each loop
      System.out.println("Before studying");
      for (Student07 student : students) {
         System.out.println(student);
      }

      //randomly select students to study to simulate an actual application
      for (int i = 0; i < 10; i++) {
         //pick random student
         int index = (int)(Math.random() * numberOfStudents);
         Student07 student = students[index];

         //add random studying time between 0 and 5 hours
         double time = Math.random() * 5;
         student.study(time);
      }

      //print students after studying with C-style for loop
      System.out.println("After studying");
      for (int i = 0; i < students.length; i++) {
         System.out.println(students[i]);
      }
   }
```

Output

```
Before studying
Name: Alice, graduation year: 2027
        Hours studying: 0.0
        Hours in class: 0.0
Name: Bob, graduation year: 2024
        Hours studying: 0.0
        Hours in class: 0.0
Name: Charlie, graduation year: 2025
        Hours studying: 0.0
        Hours in class: 0.0
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Charlie. I'm studying!
Hi Mom! It's Charlie. I'm studying!
Hi Mom! It's Charlie. I'm studying!
```

Print all students using C-style loop after studying is complete

StudentTrackerApp is an application that uses the Student07 class

StudentTrackerApp.java

```
public class StudentTrackerApp {
   public static void main(String[] args) {
      int numberOfStudents = 3;
      Student07[] students = new Student07[numberOfStudents];
      students[0] = new Student07("Alice", 2027);
      students[1] = new Student07("Bob", 2024);
      students[2] = new Student07("Charlie", 2025);

      //print students with for-each loop
      System.out.println("Before studying");
      for (Student07 student : students) {
         System.out.println(student);
      }

      //randomly select students to study to simulate an actual application
      for (int i = 0; i < 10; i++) {
         //pick random student
         int index = (int)(Math.random() * numberOfStudents);
         Student07 student = students[index];

         //add random studying time between 0 and 5 hours
         double time = Math.random() * 5;
         student.study(time);
      }

      //print students after studying with C-style for loop
      System.out.println("After studying");
      for (int i = 0; i < students.length; i++) {
         System.out.println(students[i]);
      }
   }
```

Output

```
Before studying
Name: Alice, graduation year: 2027
        Hours studying: 0.0
        Hours in class: 0.0
Name: Bob, graduation year: 2024
        Hours studying: 0.0
        Hours in class: 0.0
Name: Charlie, graduation year: 2025
        Hours studying: 0.0
        Hours in class: 0.0
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Alice. I'm studying!
Hi Mom! It's Bob. I'm studying!
Hi Mom! It's Charlie. I'm studying!
Hi Mom! It's Charlie. I'm studying!
Hi Mom! It's Charlie. I'm studying!
After studying
Name: Alice, graduation year: 2027
        Hours studying: 7.613054664089111
        Hours in class: 0.0
Name: Bob, graduation year: 2024
        Hours studying: 7.355890449244431
        Hours in class: 0.0
Name: Charlie, graduation year: 2025
        Hours studying: 8.283218705770441
        Hours in class: 0.0
```

Will the output be the same if this application program is run again?
No!

Key points

1. Encapsulation brings code and data together into one thing called an object
2. A class provides a blueprint for instantiating (creating) objects
3. An object's data is stored in instance variables
4. Many objects can be instantiated from a class, each object gets its own instance variables
5. Data can be public or private
6. Access to data is normally controlled by getter (return varible's value) and setter (update variable's value) methods
7. Constructors are a way to initialize new objects
8. Constructors are called when an object is first instantiated
9. Primitive variables are stored on the stack
10. Objects are stored in the heap
11. Multiple applications can use the same class