

CS 10:


Problem solving via Object Oriented
Programming

Encapsulation

Today main learning objectives

1. Create a class applying the encapsulation concept
2. Write instance variables
3. Write methods
4. Define difference between objects and primitives
5. Define difference between applications and classes

Agenda

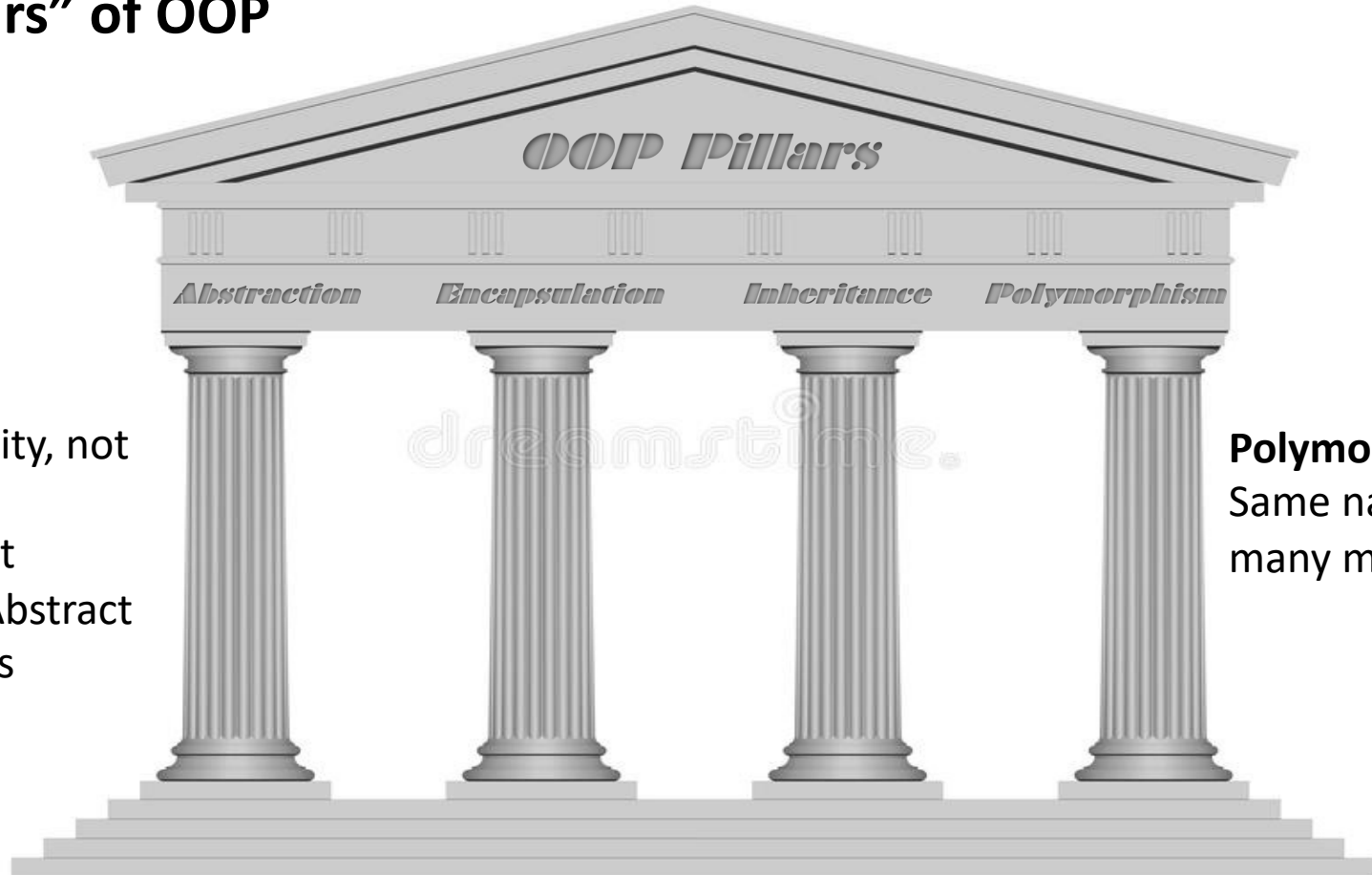
- 
1. Encapsulation
 2. Getters/setters
 3. Constructors
 4. Objects vs. primitives
 5. Applications vs classes

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

Four “pillars” of OOP

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

We start with a Student class to represent one student



We start with a Student class to represent one student

Student class



Data

- ID
- Name
- Year

Methods (code)

- Study
- Attend class

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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */  
  
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", Name: " + alice.name +  
            ", Year: " + alice.graduationYear);  
    }  
}
```


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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

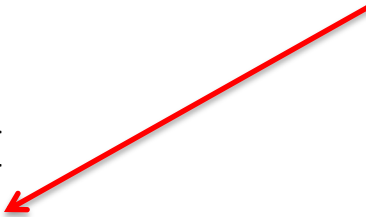
```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", Name: " + alice.name +  
            ", Year: " + alice.graduationYear);  
    }  
}
```

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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", Name: " + alice.name +  
            ", Year: " + alice.graduationYear);  
    }  
}
```

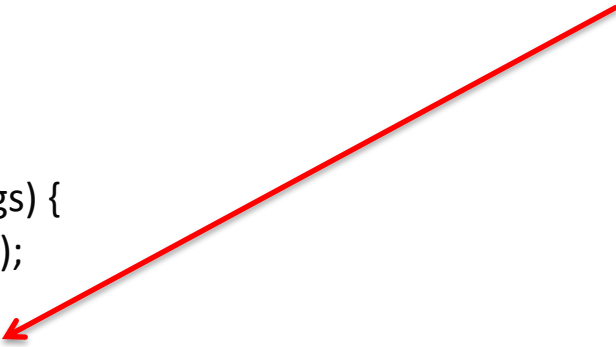


Instance variables *can* be accessed using the dot operator


Student0.java

```
/**  
 * Student series demonstrates encapsulation by representing a student in a class  
 * Student0 - base example with student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", Name: " + alice.name +  
            ", Year: " + alice.graduationYear);  
    }  
}
```



Agenda

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

Access modifiers such as “protected” can prevent outside modification

Student01.java

```
public class Student01 {  
    protected String studentId;  
    protected String name;  
    protected int graduationYear;  
  
    /**  
     * Setters for instance variables  
     */  
    public void setId(String studentId) { this.studentId = studentId; }  
    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.setId("f00xyz");  
        alice.setName("Alice");  
        alice.setYear(2027);  
        System.out.println("ID: " + alice.studentId +  
            ", Name: " + alice.name +  
            ", Year: " + alice.graduationYear);  
    }  
}
```


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

Student01.java

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

    /**
     * Setters for instance variables
     */
    public void setId(String studentId) { this.studentId = studentId; }
    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.setId("f00xyz");
        alice.setName("Alice");
        alice.setYear(2027);
        System.out.println("ID: " + alice.studentId +
            ", Name: " + alice.name +
            ", Year: " + alice.graduationYear);
    }
}
```



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


Student01.java

```
public class Student01 {  
    protected String studentId; ←  
    protected String name; ←  
    protected int graduationYear; ←  
  
    /**  
     * Setters for instance variables  
     */  
    public void setId(String studentId) { this.studentId = studentId; }  
    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.setId("f00xyz");  
        alice.setName("Alice");  
        alice.setYear(2027);  
        System.out.println("ID: " + alice.studentId +  
            ", Name: " + alice.name +  
            ", Year: " + alice.graduationYear);  
    }  
}
```

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

Student02.java

```
public class Student02 {  
    protected String studentId;  
    protected String name;  
    protected int graduationYear;  
  
    /**  
     * Setters for instance variables  
     */  
    public void setId(String studentId) { this.studentId = studentId; }  
    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>

“Getters” return values

Student03.java

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

<snip>

```
/**
```

```
 * Getters for instance variables
```

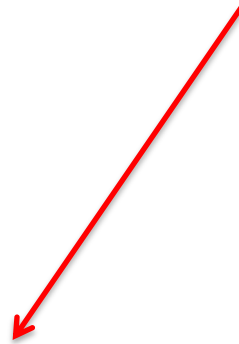
```
 */
```

```
public String getId() { return studentId; }
```

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

```
public int getGraduationYear() { return graduationYear; }
```

```
public static void main(String[] args) {  
    Student03 alice = new Student03();  
    alice.setId("f00xyz");  
    alice.setName("Alice");  
    alice.setYear(2027);
```



Agenda

1. Encapsulation

2. Getters/setters

 3. Constructors

4. Objects vs. primitives

5. Applications vs classes

Constructors allow us to initialize an object when it is instantiated

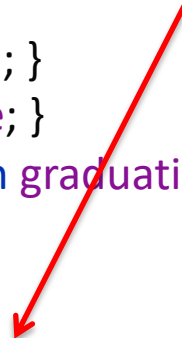
Student03.java

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

<snip>

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


```
public static void main(String[] args) {  
    Student03 alice = new Student03();  
    alice.setId("f00xyz");  
    alice.setName("Alice");  
    alice.setYear(2027);
```



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

Student04.java

```
public class Student04 {  
    protected String studentId;  
    protected String name;  
    protected int graduationYear;  
  
    public Student04() {  
        //default constructor: you get this by default  
    }  
  
    public Student04(String studentId, String name, int year) {  
        this.studentId = studentId;  
        this.name = name;  
        graduationYear = year;  
    }  
}
```




<snip>

```
public static void main(String[] args) {  
    Student04 abby = new Student04();  
    Student04 alice = new Student04("f00xyz", "Alice", 2027);  
}
```

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

Student04.java

```
public class Student04 {  
    protected String studentId;  
    protected String name;  
    protected int graduationYear;  
  
    public Student04() {  
        //default constructor: you get this by default  
    }  
  
    public Student04(String studentId, String name, int year) {  
        this.studentId = studentId;  
        this.name = name;  
        graduationYear = year;  
    }  
}
```




<snip>

```
public static void main(String[] args) {  
    Student04 abby = new Student04();  
    Student04 alice = new Student04("f00xyz", "Alice", 2027);  
}
```

Objects can have other methods other than a constructor

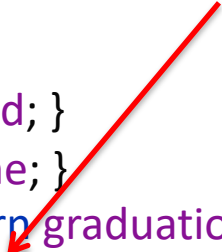
Student05.java

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



<snip>

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




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;  
}
```

Agenda

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

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

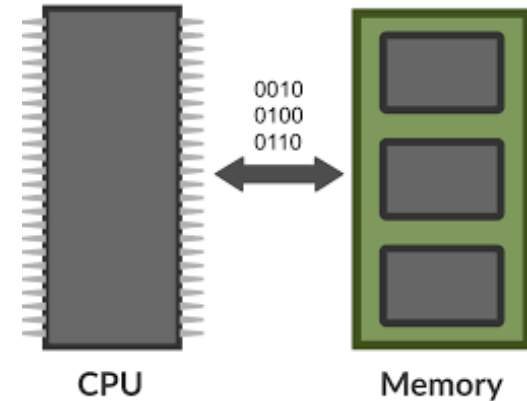
Common primitive types

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)

Memory used by programs

- Any program requires some memory to store data that will be processed
 - Note that memory is limited; e.g., RAM 8GB, 16GB, ...



Source: Khan Academy

Can we *always* know how much memory a program requires before running it?

Memory used by programs

- **Stack**: fixed-size segment of memory where data (e.g., program's local variables) get added and/or removed in a last-in-first-out (LIFO) manner
- **Heap**: segment of memory for dynamic memory allocation (take CS50 for more)

e.g., from Mac's terminal,
command to see user's limit

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size          (kbytes, -d) unlimited
file size              (blocks, -f) unlimited
max locked memory      (kbytes, -l) unlimited
max memory size        (kbytes, -m) unlimited
open files              (-n) 256
pipe size              (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes     (-u) 2784
virtual memory          (kbytes, -v) unlimited
```

Java defines a number of 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

Stack

Heap

```
6 public class MemoryAllocationPrimitives {
7
8     public static void main(String[] args) {
9         //declare local variables
10        int i; double d; boolean b; char c;
11
12        //assign values to local variables
13        i=7; d=1.6; b=true; c='a';
14
15        //print new values
16        System.out.println("Local variables: "+
17            "i="+i+" d="+d+" b="+b+" c="+c);
18    }
19 }
```

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy
terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom

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

Stack

Heap

```
6 public class MemoryAllocationPrimitives {
7
8     public static void main(String[] args) {
9         //declare local variables
10        int i; double d; boolean b; char c;
11
12        //assign values to local variables
13        i=7; d=1.6; b=true; c='a';
14
15        //print new values
16        System.out.println("Local variables: "+
17            "i="+i+" d="+d+" b="+b+" c="+c);
18    }
19 }
```



i 

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy
terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom

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

Stack

Heap

```
6 public class MemoryAllocationPrimitives {
7
8     public static void main(String[] args) {
9         //declare local variables
10        int i; double d; boolean b; char c;
11
12        //assign values to local variables
13        i=7; d=1.6; b=true; c='a';
14
15        //print new values
16        System.out.println("Local variables: "+
17            "i="+i+" d="+d+" b="+b+" c="+c);
18    }
19 }
```



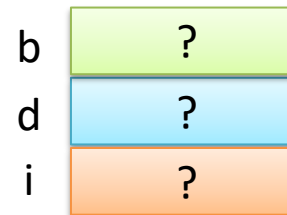
Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy
terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom

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

```
6 public class MemoryAllocationPrimitives {
7
8     public static void main(String[] args) {
9         //declare local variables
10        int i; double d; boolean b; char c;
11
12        //assign values to local variables
13        i=7; d=1.6; b=true; c='a';
14
15        //print new values
16        System.out.println("Local variables: "+
17            "i="+i+" d="+d+" b="+b+" c="+c);
18    }
19 }
```

Stack

Heap



Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy
terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom

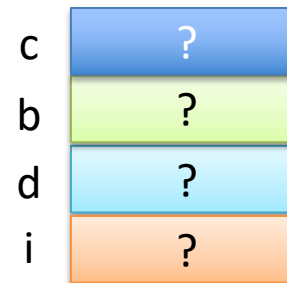
Declaring a primitive variable allocates stack space that holds variable's value

```
6 public class MemoryAllocationPrimitives {
7
8     public static void main(String[] args) {
9         //declare local variables
10        int i; double d; boolean b; char c;
11
12        //assign values to local variables
13        i=7; d=1.6; b=true; c='a';
14
15        //print new values
16        System.out.println("Local variables: "+
17            "i="+i+" d="+d+" b="+b+" c="+c);
18    }
19 }
```



Stack

Heap



Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy
terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom

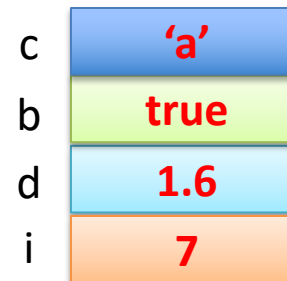
Declaring a primitive variable allocates stack space that holds variable's value

```
6 public class MemoryAllocationPrimitives {
7
8     public static void main(String[] args) {
9         //declare local variables
10        int i; double d; boolean b; char c;
11
12        //assign values to local variables
13        i=7; d=1.6; b=true; c='a';
14
15        //print new values
16        System.out.println("Local variables: "+
17            "i="+i+" d="+d+" b="+b+" c="+c);
18    }
19 }
```



Stack

Heap



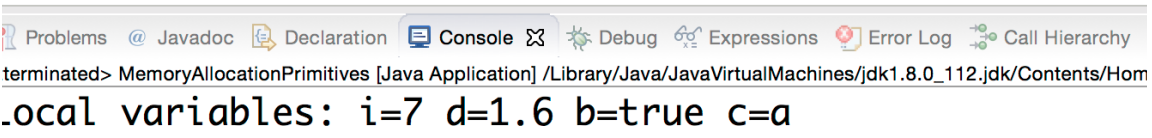
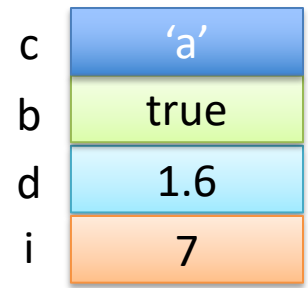
Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy
terminated> MemoryAllocationPrimitives [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Hom

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

```
6 public class MemoryAllocationPrimitives {
7
8     public static void main(String[] args) {
9         //declare local variables
10        int i; double d; boolean b; char c;
11
12        //assign values to local variables
13        i=7; d=1.6; b=true; c='a';
14
15        //print new values
16        System.out.println("Local variables: "+
17            "i="+i+" d="+d+" b="+b+" c="+c);
18    }
19 }
```



Stack **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("f00xyz", "Alice", 2027);  
}
```

Stack

Heap

alice

6acbcfc0

Memory location
6acbcfc0

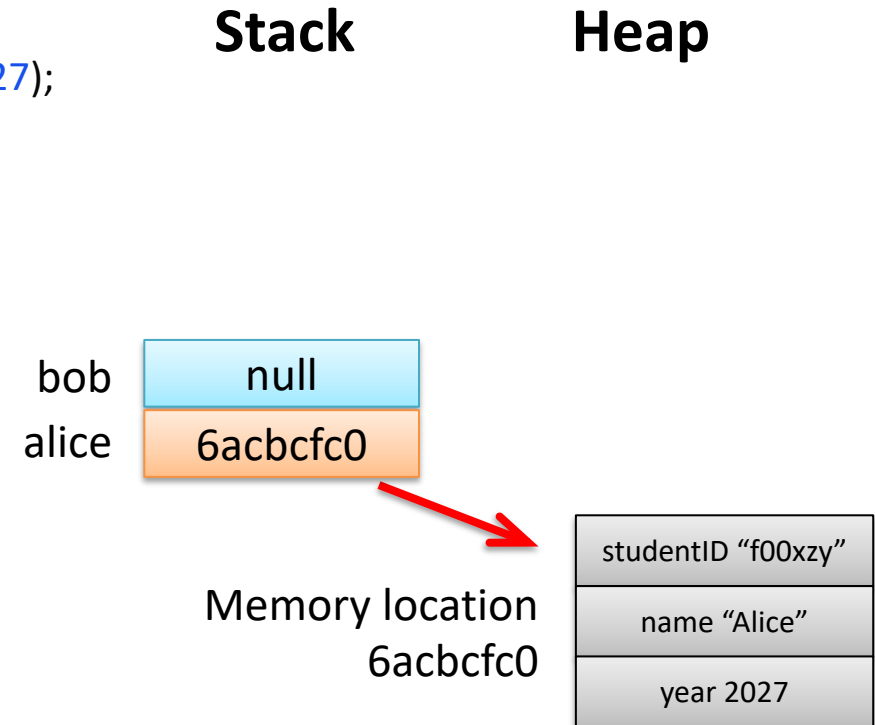
studentID "f00xyz"

name "Alice"

year 2027

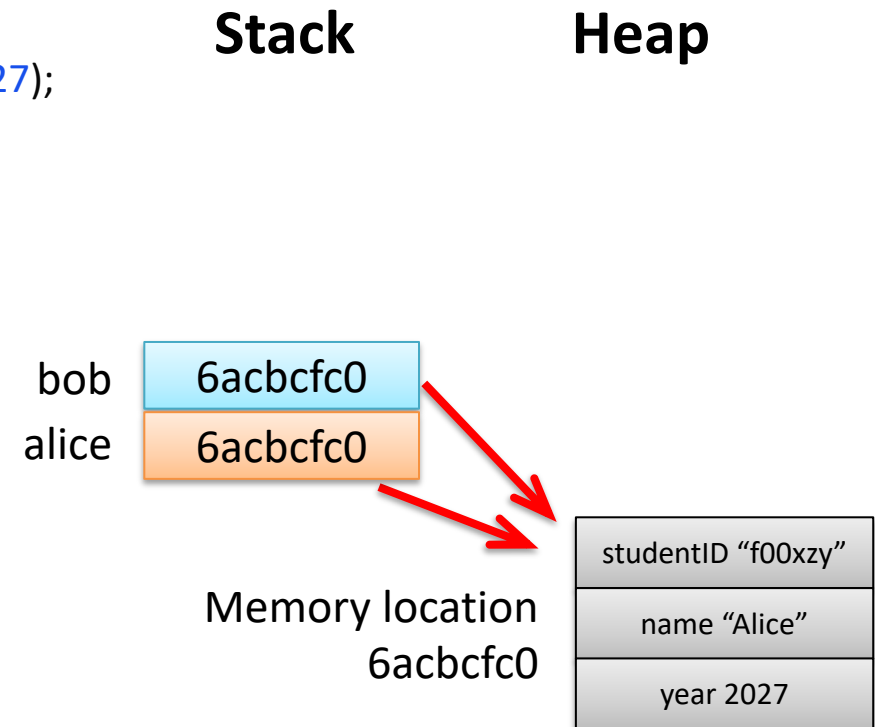
Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {  
    //declare Student objects  
    Student05 alice = new Student05("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword
```



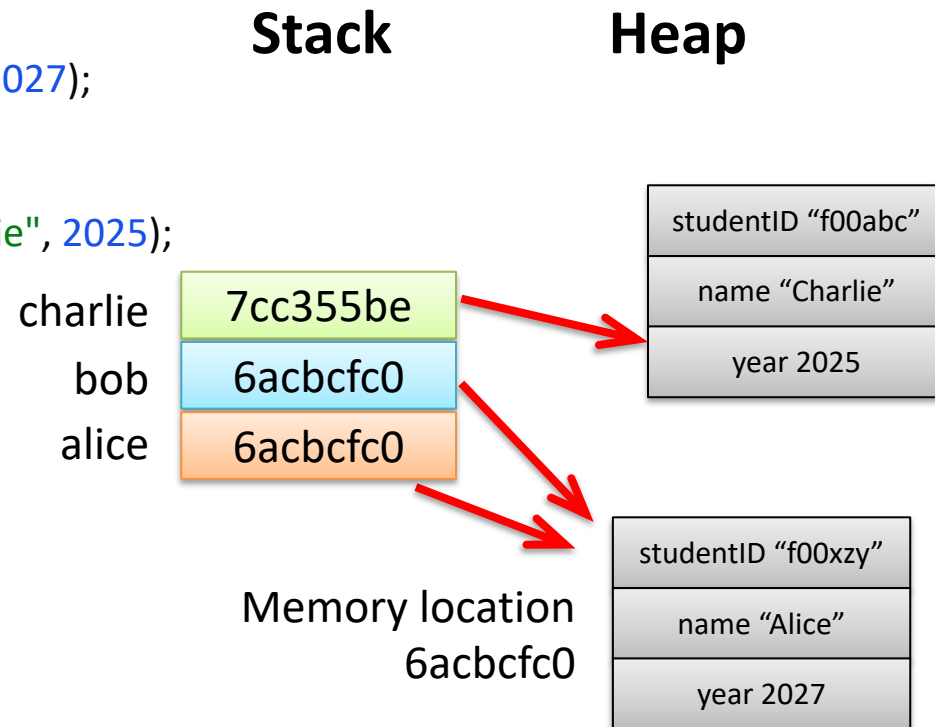
Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {  
    //declare Student objects  
    Student05 alice = new Student05("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
}
```



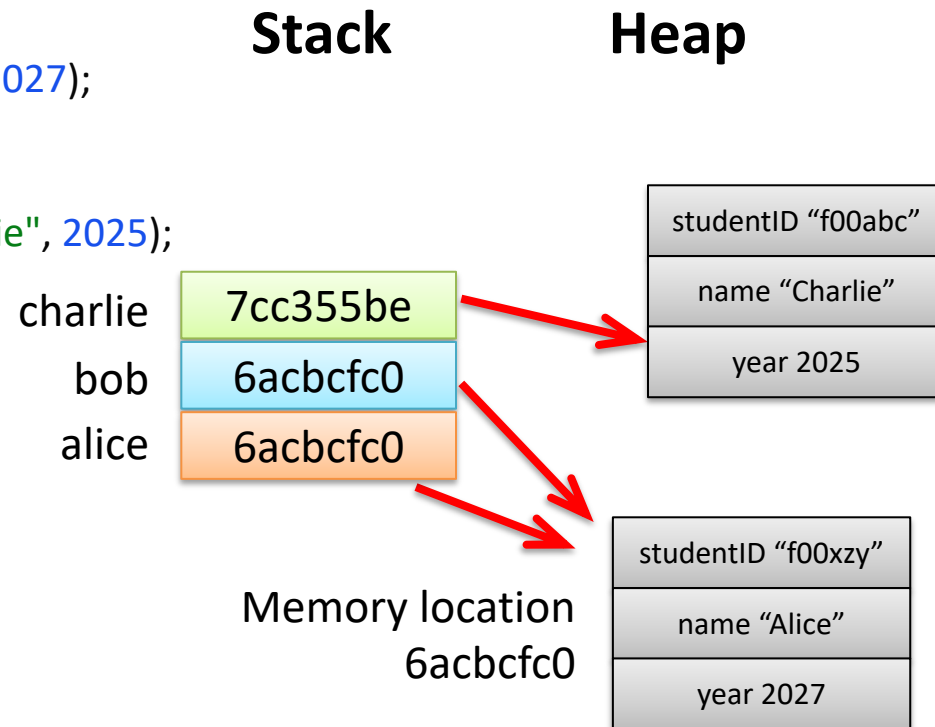
Declaring objects makes pointer on the stack, but object itself is elsewhere

```
public static void main(String[] args) {  
    //declare Student objects  
    Student05 alice = new Student05("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "Charlie", 2025);  
}
```



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

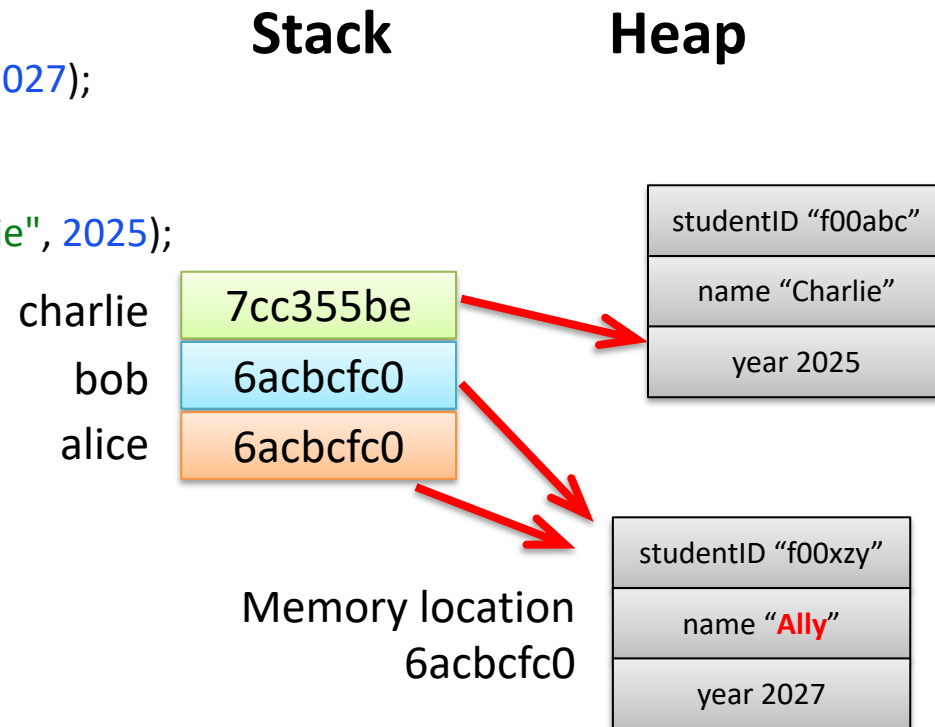
```
public static void main(String[] args) {  
    //declare Student objects  
    Student05 alice = new Student05("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "Charlie", 2025);  
    System.out.println(alice.name + " " + bob.name);  
}
```



Output
Alice Alice

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

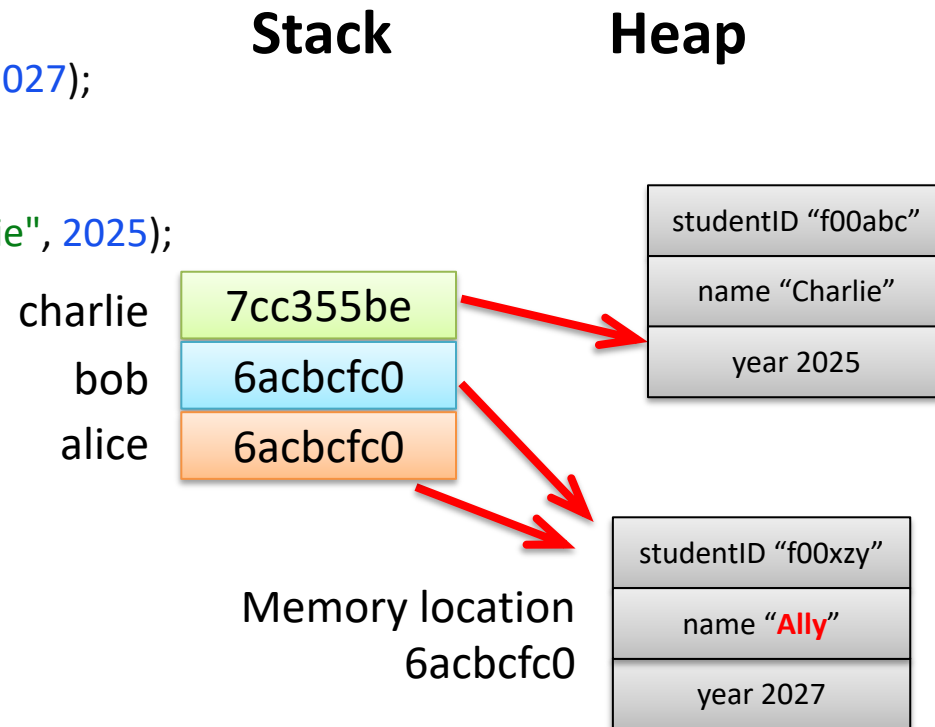
```
public static void main(String[] args) {  
    //declare Student objects  
    Student05 alice = new Student05("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "Charlie", 2025);  
    System.out.println(alice.name + " " + bob.name);  
  
    //update alice's name  
    alice.setName("Ally");  
}
```



Output
Alice Alice

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("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "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
```

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

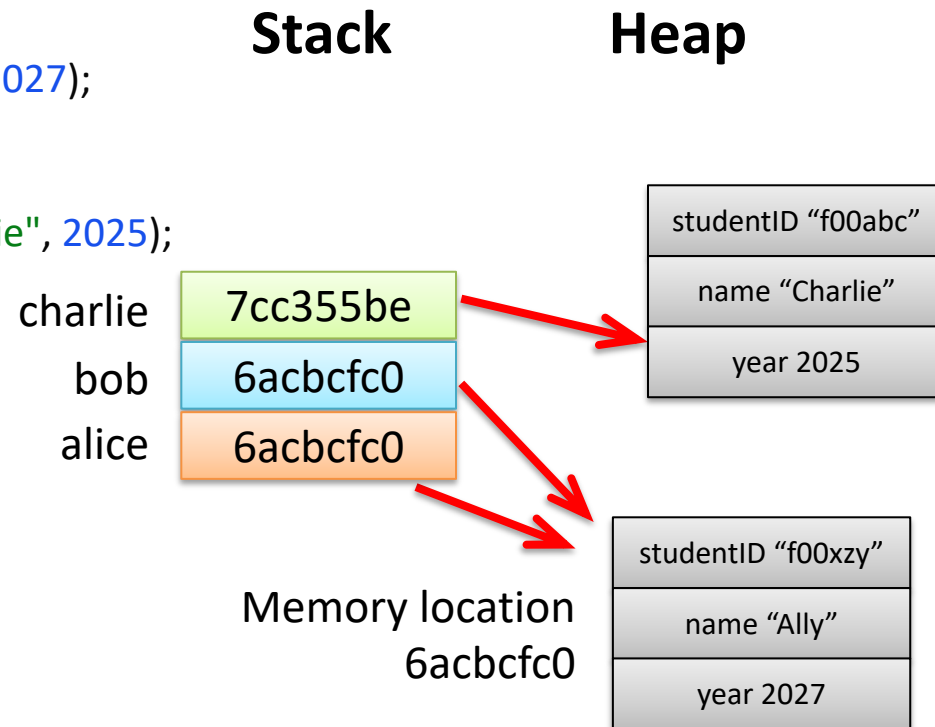
```
public static void main(String[] args) {  
    //declare Student objects  
    Student05 alice = new Student05("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "Charlie", 2025);  
    System.out.println(alice.name + " " + 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



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 = "Student ID: " + studentId + ", 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("f00xyz", "Alice", 2027); //calls first constructor
    alice.study(1.5);
    alice.attendClass(1.1);
    System.out.println(alice);
}
```

Output

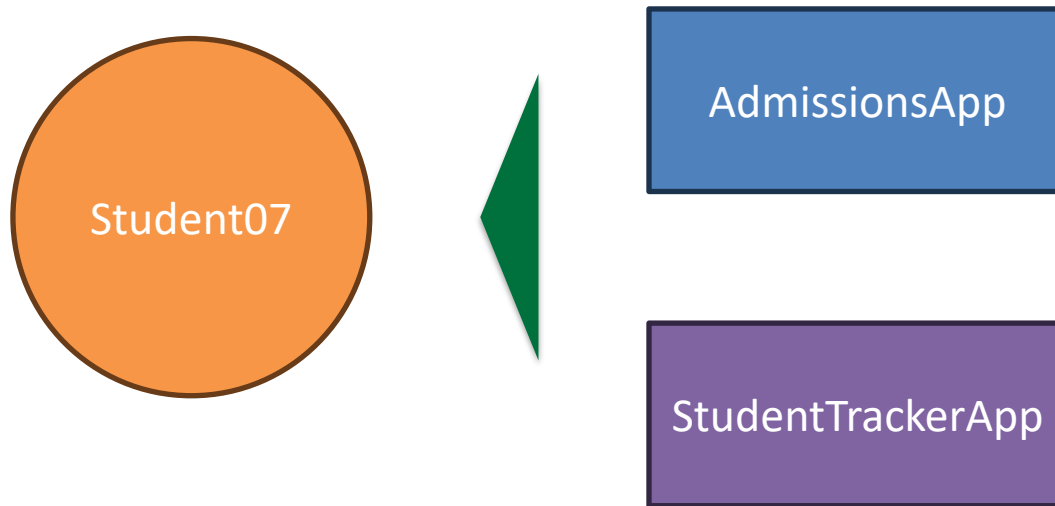
```
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 1.5
    Hours in class: 1.1
```

**DO NOT print in
toString method!**

Agenda

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

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



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

Project ▾

cs10a ~/Documents/IdeaProjects/cs10a

- ▾ .idea
- ▾ data
- ▾ day1
- ▾ day2
 - MemoryAllocationObjects
 - MemoryAllocationPrimitives
 - Student0
 - Student01
 - Student02
 - Student03
 - Student04
 - Student05
 - Student06
 - Student07
 - StudentTrackerApp

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

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

    public Student07(String studentId, String name, int year) {
        this.studentId = studentId;
        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 = "Student ID: " + studentId + ", name: " + name + ", graduation year: " + graduationYear + "\n";
        s += "\tHours studying: " + studyHours + "\n";
        s += "\tHours in class: " + classHours;
        return s;
    }
}
```

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("f00xyz", "Alice", 2027);  
        students[1] = new Student07("f00123", "Bob", 2024);  
        students[2] = new Student07("f00abc", "Charlie", 2025);  
  
        //print students with for-each loop  
        System.out.println("Before studying");  
        for (Student07 student : students) {  
            System.out.println(student);  
        }  
    }  
}
```

StudentTrackerApp.java

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("f00xyz", "Alice", 2027);  
        students[1] = new Student07("f00123", "Bob", 2024);  
        students[2] = new Student07("f00abc", "Charlie", 2025);  
  
        //print students with for-each loop  
        System.out.println("Before studying");  
        for (Student07 student : students) {  
            System.out.println(student);  
        }  
    }  
}
```

Output

```
Before studying  
Student ID: f00xyz, name: Alice, graduation year: 2027  
    Hours studying: 0.0  
    Hours in class: 0.0  
Student ID: f00123, name: Bob, graduation year: 2024  
    Hours studying: 0.0  
    Hours in class: 0.0  
Student ID: f00abc, name: Charlie, graduation year: 2025  
    Hours studying: 0.0  
    Hours in class: 0.0
```

StudentTrackerApp.java

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("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, name: Charlie, graduation year: 2025
    Hours studying: 0.0
    Hours in class: 0.0
```

StudentTrackerApp.java

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("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, 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.java

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("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, 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.java

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("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, 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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 7.613054664089111
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 7.355890449244431
    Hours in class: 0.0
Student ID: f00abc, name: Charlie, graduation year: 2025
    Hours studying: 8.283218705770441
    Hours in class: 0.0
```

StudentTrackerApp.java

Summary

- Applying the encapsulation concept means designing a class and identifying what data and methods should be part of it
 - Instance variables for data
 - protected to be only accessible by the class/subclass methods
 - Methods
 - Getters/setters for accessing/modifying data
 - Constructors to initialize an object with specific values
 - Method overloading is possible
 - Other methods
- Object vs. primitives
 - Heap vs. stack
 - Reference vs. value
- Applications vs. classes
 - Classes used in application or driver programs

Next

- Inheritance

Additional Resources

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 ID, name, 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

- ID
- Name
- Year

Methods (code)

- Study
- Attend class

Student class

- We will model students as objects
- Objects encapsulate:
 - Data about a student (e.g., student ID, name)
 - 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)
 - 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)

Student0.java

ANNOTATED SLIDES

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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", 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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", 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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", 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)
- Track student's ID, name, and graduation year
- Must declare data type (two Strings and one integer here)
- Java initializes instance variables 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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", 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 studentID and 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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", Name: " + alice.name +  
            ", Year: " + alice.graduationYear);  
    }  
}
```

- Alice's instance variables set by using the dot operator
- For example: *alice.studentId*
- Alice's details 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 student ID, name, and graduation year  
 *  
 * @author Tim Pierson, Dartmouth CS10, Winter 2024  
 */
```

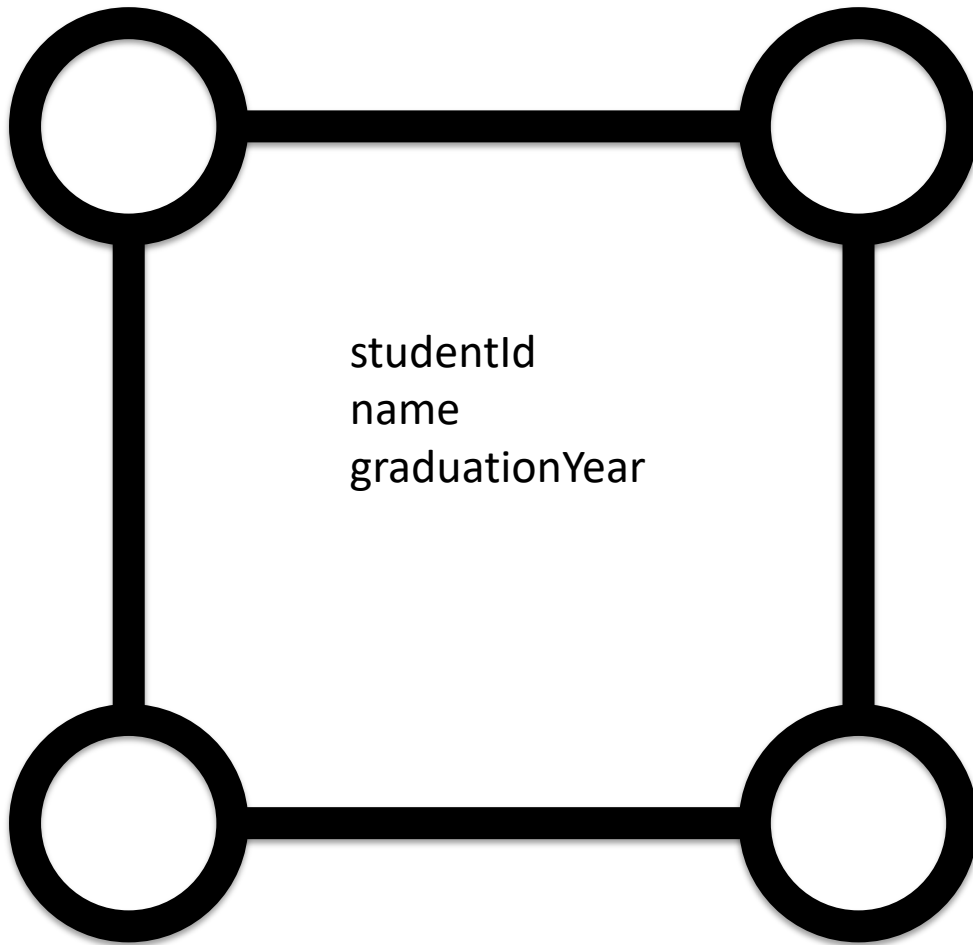
```
public class Student0 {  
    String studentId;  
    String name;  
    int graduationYear;  
  
    public static void main(String[] args) {  
        Student0 alice = new Student0();  
        alice.studentId = "f00xyz";  
        alice.name = "Alice";  
        alice.graduationYear = 2027;  
        System.out.println("ID: " + alice.studentId +  
            ", 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

Getters/setters

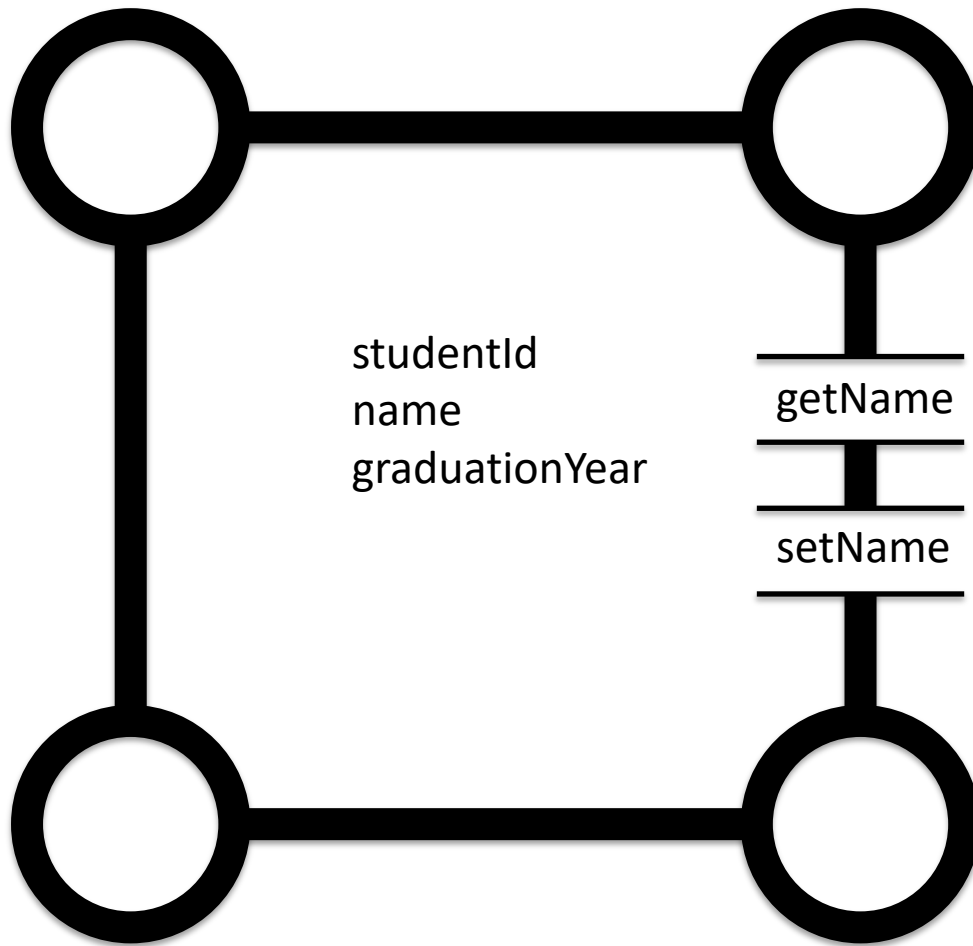
ANNOTATED SLIDES

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 studentId;  
    protected String name;  
    protected int 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

```
/**
```

```
 * Setters for instance variables
```

```
 */
```

```
public void setId(String studentId) { this.studentId = studentId; }  
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.setId("f00xyz");  
    alice.setName("Alice");  
    alice.setYear(2027);  
    System.out.println("ID: " + alice.studentId +  
        ", Name: " + alice.name +  
        ", Year: " + alice.graduationYear);
```

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

Student01.java

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

    /**
     * Setters for instance variables
     */

    public void setId(String studentId) { this.studentId = studentId; }
    public void setName(String name) { this.name = name; }
    public void setYear(int year) { graduationYear = year; }
```

- “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

```
public static void main(String[] args) {
    Student01 alice = new Student01();
    alice.setId("f00xyz");
    alice.setName("Alice");
    alice.setYear(2027);
    System.out.println("ID: " + alice.studentId +
        ", Name: " + alice.name +
        ", Year: " + alice.graduationYear);
}
```

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

Student01.java

```
public class Student01 {  
    protected String studentId;  
    protected String name;  
    protected int graduationYear;  
  
    /**  
     * Setters for instance variables  
     */  
    public void setId(String studentId) { this.studentId = studentId; }  
    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.setId("f00xyz");  
    alice.setName("Alice");  
    alice.setYear(2027);  
    System.out.println("ID: " + alice.studentId +  
        ", 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 studentId;
    protected String name;
    protected int graduationYear;

    /**
     * Setters for instance variables
     */

    public void setId(String studentId) { this.studentId = studentId; }
    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.setId("f00xyz");
        alice.setName("Alice");
        alice.setYear(2027);
        System.out.println("ID: " + alice.studentId +
            ", Name: " + alice.name +
            ", Year: " + alice.graduationYear);
    }
}
```

Instance variable *studentID* is a String, so is setter parameter *studentID*

- “*this.studentId*” means the “instance variable *studentId*” for this object
- *studentId* 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 studentId;  
    protected String name;  
    protected int graduationYear;  
  
    /**  
     * Setters for instance variables  
     */  
    public void setId(String studentId) { this.studentId = studentId; }  
    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.setId("f00xyz");  
        alice.setName("Alice");  
        alice.setYear(2027);  
        System.out.println("ID: " + alice.studentId +  
            ", 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 studentId;
    protected String name;
    protected int graduationYear;

    /**
     * Setters for instance variables
     */
    public void setId(String studentId) { this.studentId = studentId; }
    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**

“Getters” return values

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

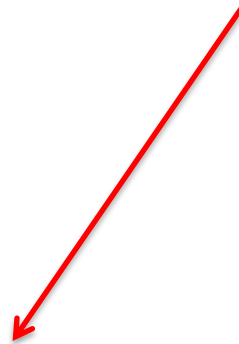
<snip>

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

```
public static void main(String[] args) {  
    Student03 alice = new Student03();  
    alice.setId("f00xyz");  
    alice.setName("Alice");  
    alice.setYear(2027);
```

Student03.java

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



Constructors

ANNOTATED SLIDES

Constructors allow us to initialize an object when it is instantiated

Student03.java

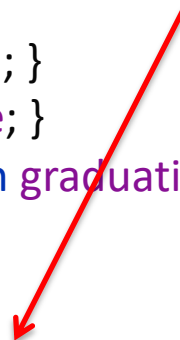
```
public class Student03 {  
    protected String studentId;  
    protected String name;  
    protected int graduationYear;
```

<snip>

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

```
public static void main(String[] args) {  
    Student03 alice = new Student03();  
    alice.setId("f00xyz");  
    alice.setName("Alice");  
    alice.setYear(2027);
```

- **Remember: by default, instance variables are initialized to 0, null, or false**
- **So, *studentId* and *name* are *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**



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

Student04.java

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

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

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

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

<snip>

```
public static void main(String[] args) {  
    Student04 abby = new Student04();  
    Student04 alice = new Student04("f00xyz", "Alice", 2027);  
}
```

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

Student04.java

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

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

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

<snip>

```
public static void main(String[] args) {  
    Student04 abby = new Student04();  
    Student04 alice = new Student04("f00xyz", "Alice", 2027);  
}
```

- This constructor takes three 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 studentId;  
    protected String name;  
    protected int graduationYear;
```

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

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

<snip>

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

```
    Student04 abby = new Student04(); //calls first constructor
```

```
    Student04 alice = new Student04("f00xyz", "Alice", 2027); //calls second constructor
```


- When *abby* is instantiated, Java calls the first constructor (no parameters provided)
- When *alice* is instantiated, Java calls the second constructor (two Strings 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 studentId;  
    protected String name;  
    protected int graduationYear;  
    protected double studyHours;  
    protected double classHours;
```

Add new instance variables to track hours studying and in class



<snip>

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

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;  
}
```

study method

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

Don't forget JavaDocs

```
/**  
 * 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;  
}
```

attendClass method
Like study method but alerts Dad

Objects vs. primitives

ANNOTATED SLIDES

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);
```

```
    }
```

```
}
```

Stack

Heap

i 

- 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
```

```
         double d; boolean b; char c;
```

```
        //assign values to local variables
```

```
        i=7; d=1.6; b=true; c='a';
```

```
        //print new values
```

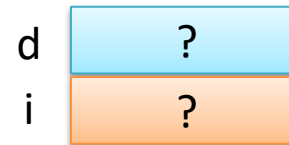
```
        System.out.println("Local variables: "+  
            "i="+i+" d="+d+" b="+b+" c="+c);
```

```
    }
```

```
}
```

Stack

Heap



- 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 boolean b; char c;
```

```
        //assign values to local variables
```

```
        i=7; d=1.6; b=true; c='a';
```

```
        //print new values
```

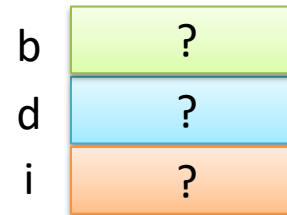
```
        System.out.println("Local variables: "+  
            "i="+i+" d="+d+" b="+b+" c="+c);
```

```
    }
```

```
}
```

Stack

Heap



- 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
```

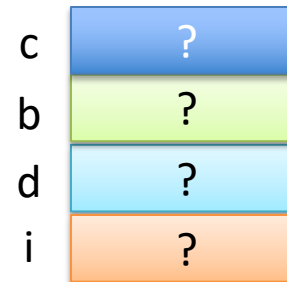
```
        System.out.println("Local variables: "+  
            "i="+i+" d="+d+" b="+b+" c="+c);
```

```
    }
```

```
}
```

Stack

Heap



- 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
```

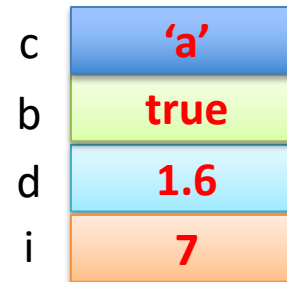
```
        System.out.println("Local variables: "+  
            "i="+i+" d="+d+" b="+b+" c="+c);
```

```
    }
```

```
}
```

Stack

Heap



- **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

Heap

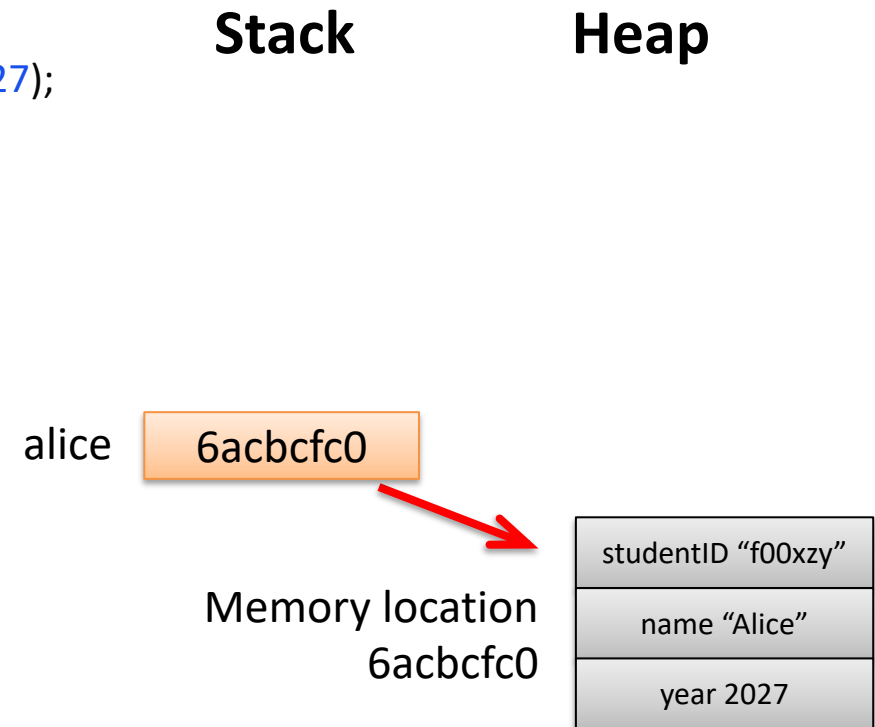
c	'a'
b	true
d	1.6
i	7

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

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("f00xyz", "Alice", 2027);  
}
```

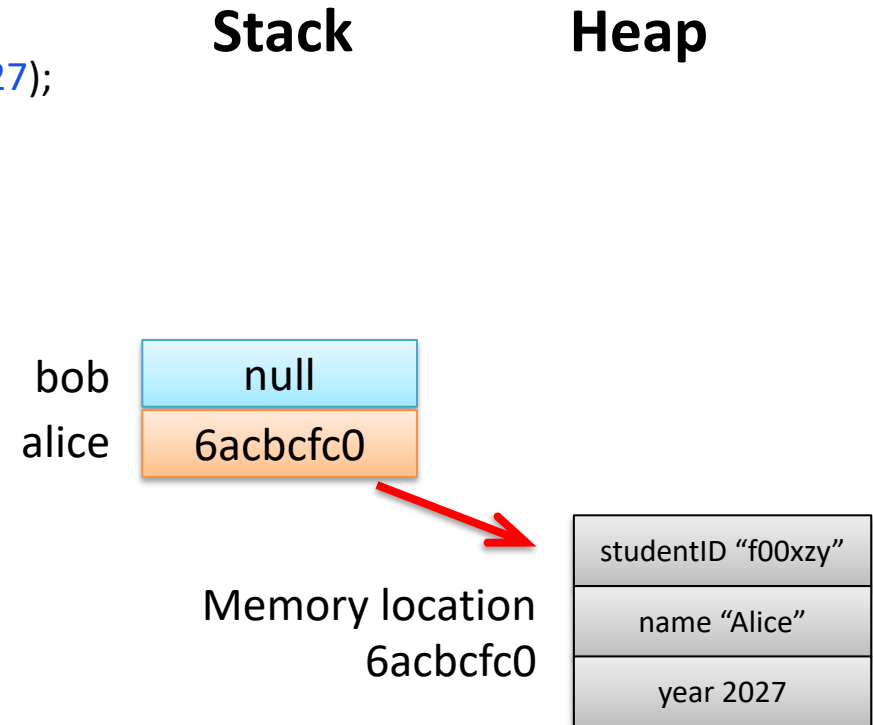
- Stack holds memory address of object (with primitives, stack holds variable's *value*)
- Memory address tells Java where to find the "alice" object in memory
- Object itself allocated elsewhere in memory (in heap, not on stack)
- OS chooses where to allocate



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("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword
```

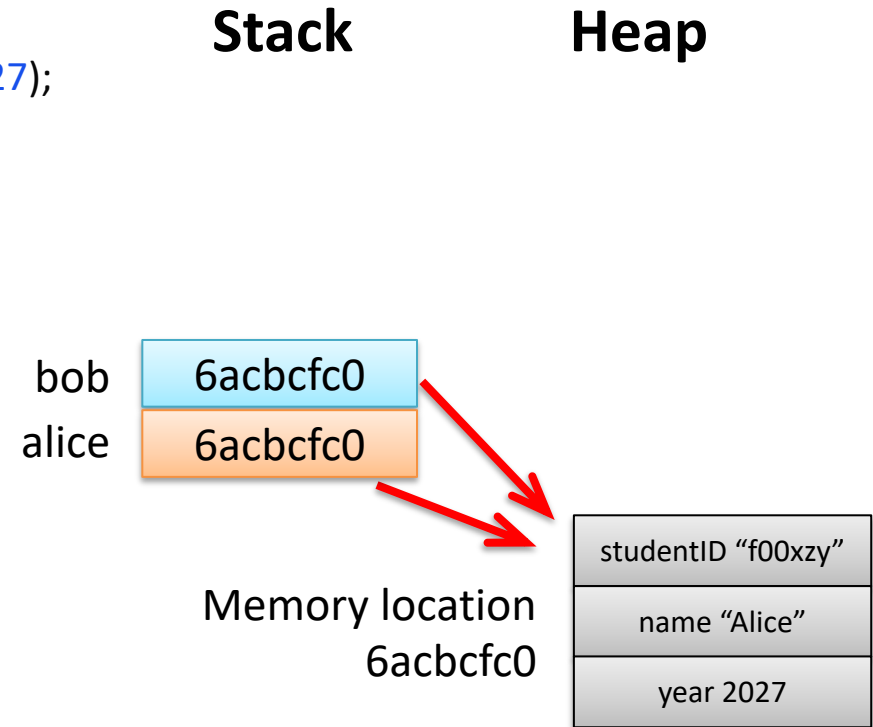
- **“bob” is allocated on the stack, but is null (points nowhere)**
- **This is because *bob* did not use the “new” keyword**
- **Null pointer exception if try to use *bob* now**



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("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
}
```

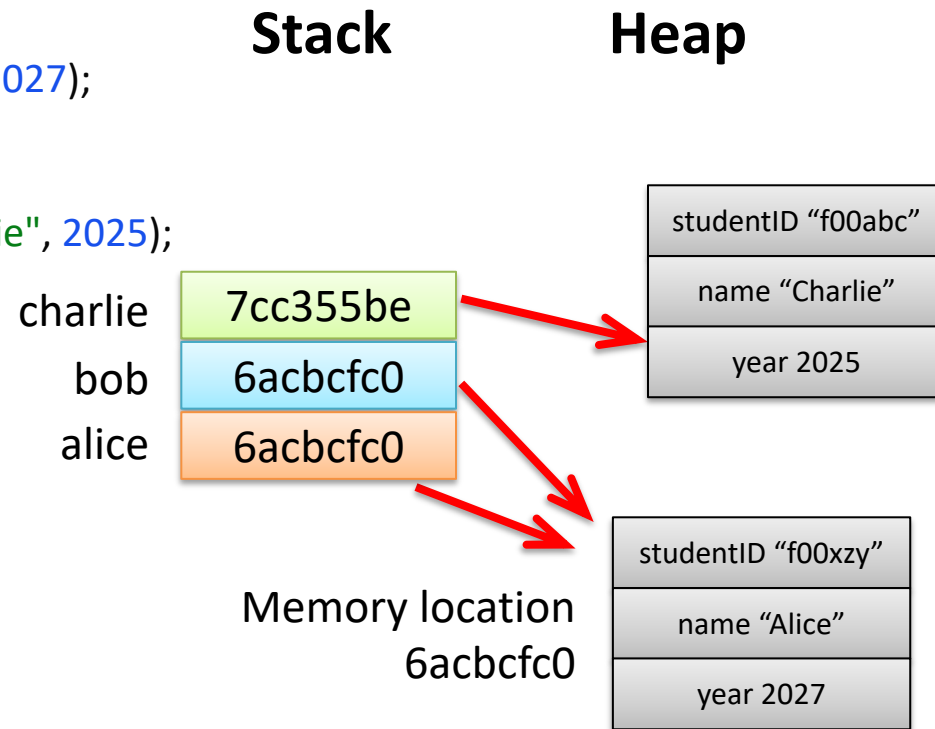
- **bob** set equal to **alice**
- **bob** gets same value on stack that **alice** holds
- **bob** now points to the exact same memory location as **alice**
- **bob** and **alice** are “aliases” of each other



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

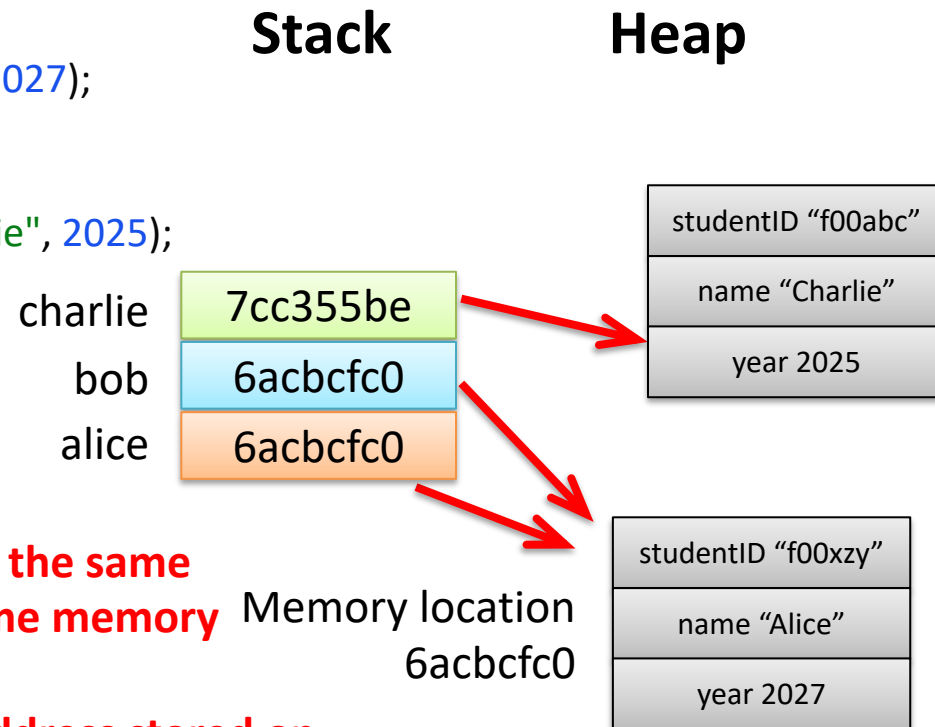
```
public static void main(String[] args) {  
    //declare Student objects  
    Student05 alice = new Student05("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "Charlie", 2025);  
}
```

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("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "Charlie", 2025);  
    System.out.println(alice.name + " " + bob.name);  
}
```



- **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**

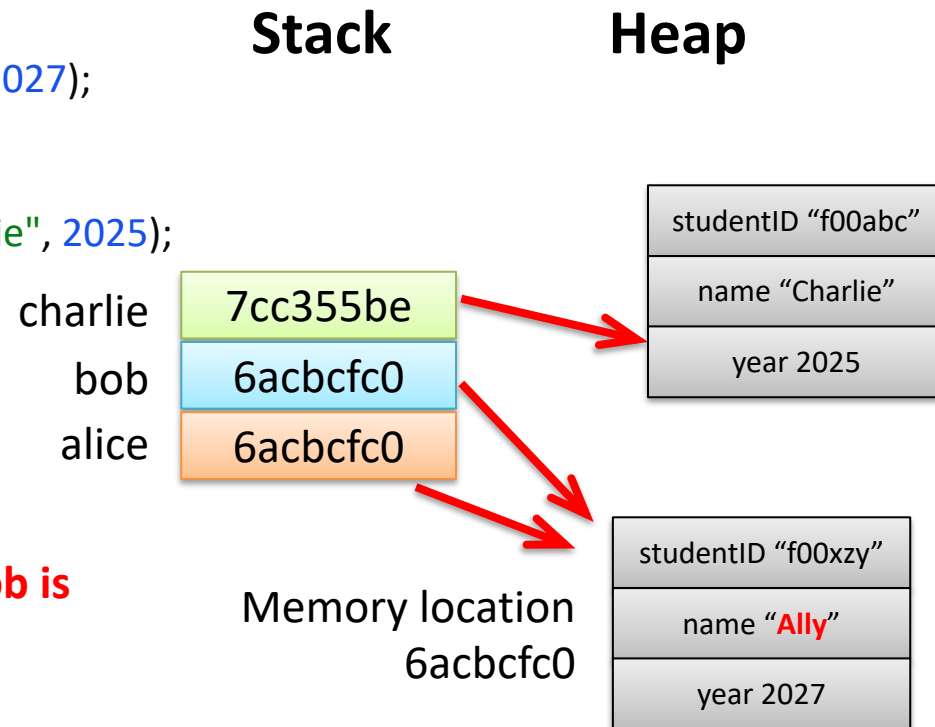
Output
Alice Alice

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("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "Charlie", 2025);  
    System.out.println(alice.name + " " + bob.name);  
  
    //update alice's name  
    alice.setName("Ally");  
}
```

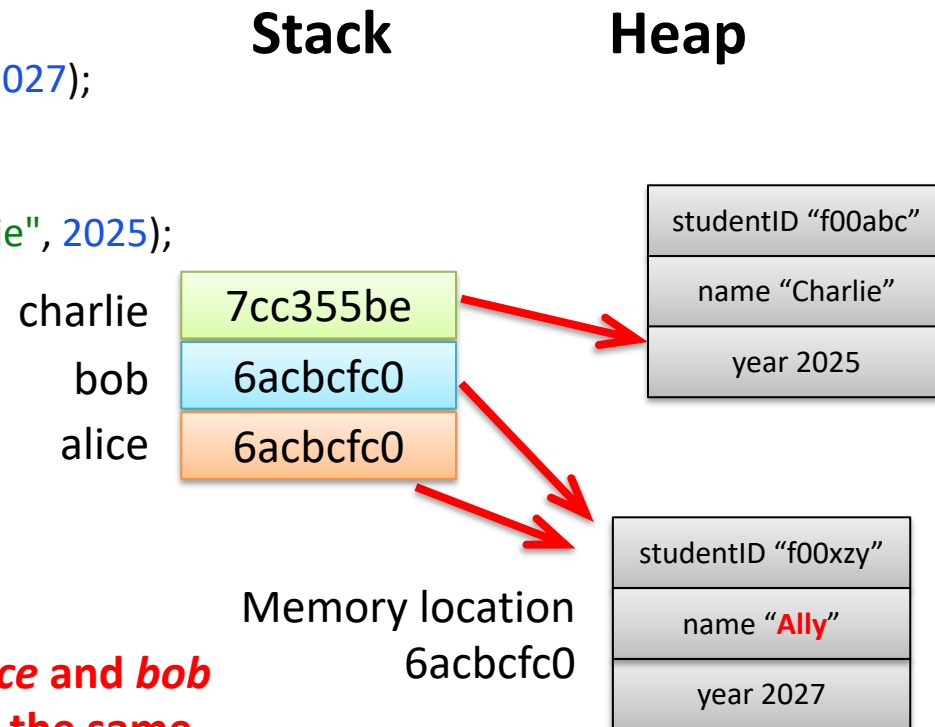
- **alice.name set to "Ally" but Bob is unchanged**
- **Expect alice's name to change**
- **What about bob?**

Output
Alice Alice



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("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "Charlie", 2025);  
    System.out.println(alice.name + " " + bob.name);  
  
    //update alice's name  
    alice.setName("Ally");  
    System.out.println(alice.name + " " + bob.name);  
}
```



- ***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"**

Output
Alice Alice
Ally Ally

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("f00xyz", "Alice", 2027);  
    Student05 bob; //notice no new keyword  
    bob = alice; //bob equals alice  
    Student05 charlie = new Student05("f00abc", "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);  
}
```

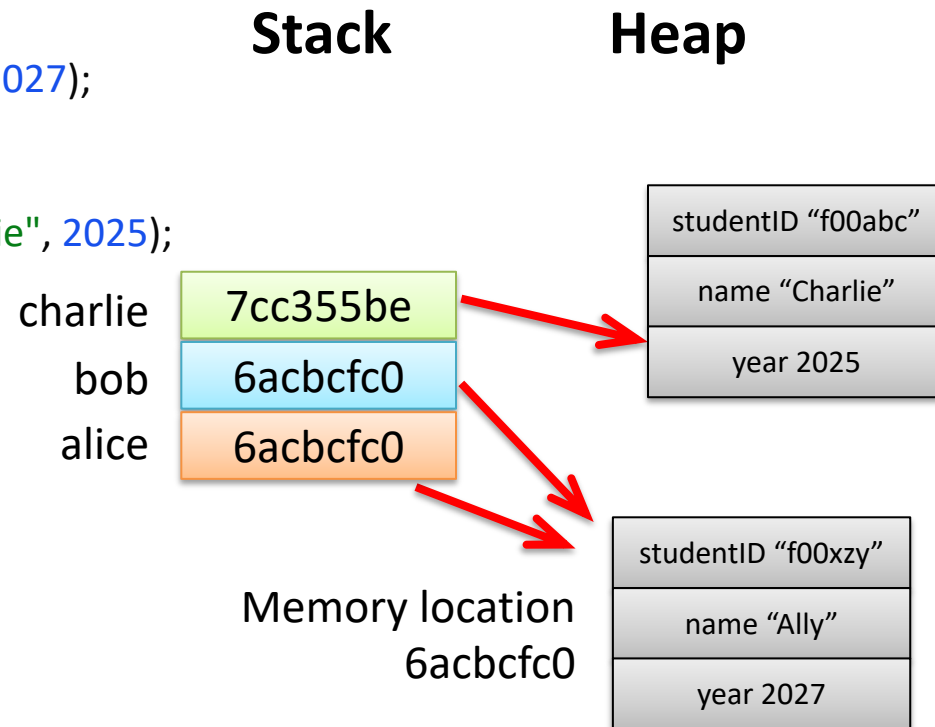
Output

Alice Alice

Ally Ally

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

- **Printing an object causes an implicit call to "toString()" function**
- **This can be overridden (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 = "Student ID: " + studentId + ", name: " + name + ", graduation year: "
        + graduationYear + "\n";
    s += "\tHours studying: " + studyHours + "\n";
    s += "\tHours in class: " + classHours;
    return s;
}
```

- ***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

```
public static void main(String[] args) {
    Student06 alice = new Student06("f00xyz", "Alice", 2027); //calls first constructor
    alice.study(1.5);
    alice.attendClass(1.1);
    System.out.println(alice);
}
```

Output

```
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 1.5
    Hours in class: 1.1
```

Tab character indents

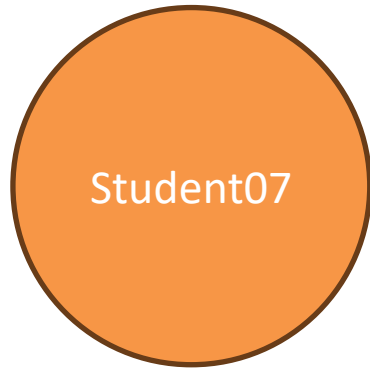
New line character puts following text on next line

DO NOT print in toString method!

Classes vs. applications

ANNOTATED SLIDES

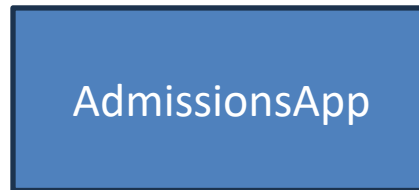
Frequently we will create classes and use them in application or driver programs



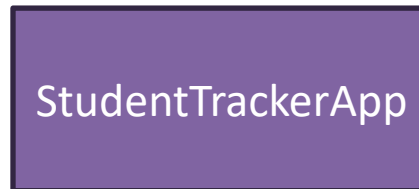
***Student07* class does not have a *main* method**

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

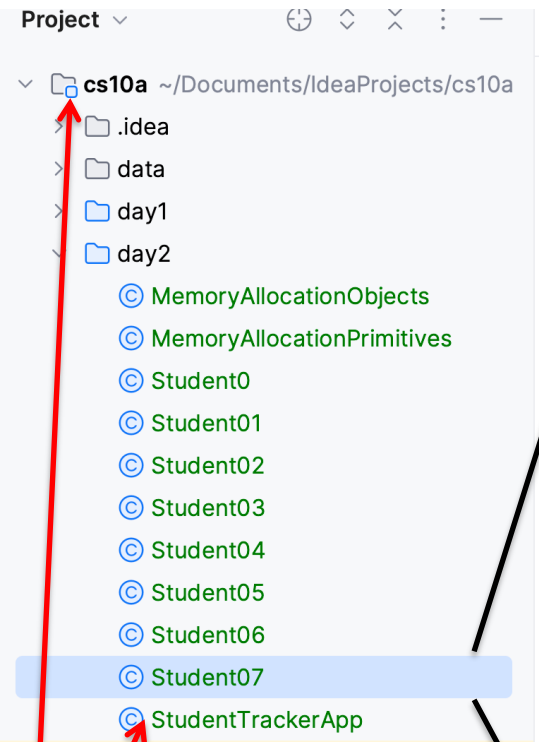


Here we have an application for a college’s Admissions office

Another application might be used by the Registrar to track students

Multiple application programs can use the Student class

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



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

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

    public Student07(String studentId, String name, int year) {
        this.studentId = studentId;
        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 = "Student ID: " + studentId + ", 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 can use Student07 class because they are both in the cs10 project (no need to import)

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("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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

```
public class StudentTrackerApp {
    public static void main(String[] args) {
        int numberOfStudents = 3;
        Student07[] students = new Student07[numberOfStudents];
        students[0] = new Student07("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "Charlie", 2025);

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

Output

```
Before studying
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, name: Charlie, graduation year: 2025
    Hours studying: 0.0
    Hours in class: 0.0
```

StudentTrackerApp.java

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("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, name: Charlie, graduation year: 2025
    Hours studying: 0.0
    Hours in class: 0.0
```

StudentTrackerApp.java

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

```
public class StudentTrackerApp {
    public static void main(String[] args) {
        int numberOfStudents = 3;
        Student07[] students = new Student07[numberOfStudents];
        students[0] = new Student07("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, 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.java

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("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, 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.java

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

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("f00xyz", "Alice", 2027);
        students[1] = new Student07("f00123", "Bob", 2024);
        students[2] = new Student07("f00abc", "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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 0.0
    Hours in class: 0.0
Student ID: f00abc, 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
Student ID: f00xyz, name: Alice, graduation year: 2027
    Hours studying: 7.613054664089111
    Hours in class: 0.0
Student ID: f00123, name: Bob, graduation year: 2024
    Hours studying: 7.355890449244431
    Hours in class: 0.0
Student ID: f00abc, name: Charlie, graduation year: 2025
    Hours studying: 8.283218705770441
    Hours in class: 0.0
```

StudentTrackerApp.java

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