Class Design Part Two

Pass by Value

static

Interfaces

Comparable

Enumerated Data Types (enum)

PASS BY VALUE

What is Stored in Memory

- Primitive variables
  - The actual value- the data
- Object variables (also called object *references*)
  - A reference/ pointer/ memory address to the place in memory where all the information about the object resides
- This is a critical distinction in Java!

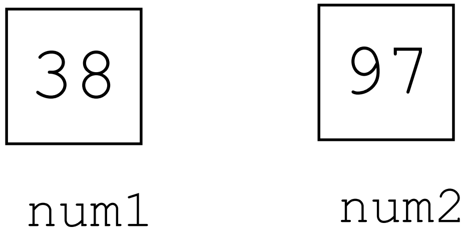
Assignment Statements

- Assignment takes the **value** on the right and stores it in the variable on the left.
- Think about what **the value** is!
    - It’s different for primitives and objects!

Assignment- Primitives

- Assignment takes the value on the right and stores it in the variable on the left.
  - For primitives, the value is just the data!

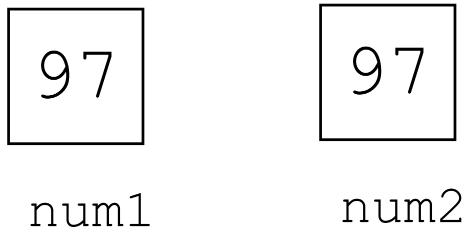
```
int num1 = 38;

int num2 = 97;
```



Assignment- Primitives

```
num1 = num2;
```

- What is the **value** of num2?
- Because it’s a primitive, the value is just the data! So the data- the actual number- is placed into num1.

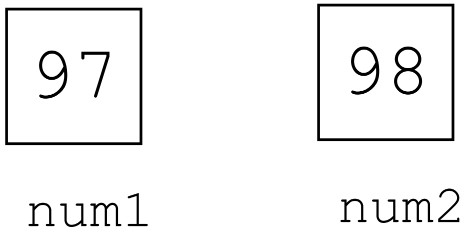


97 97

num1 num2

Assignment- Primitives

```
num2++;
```



97

num1

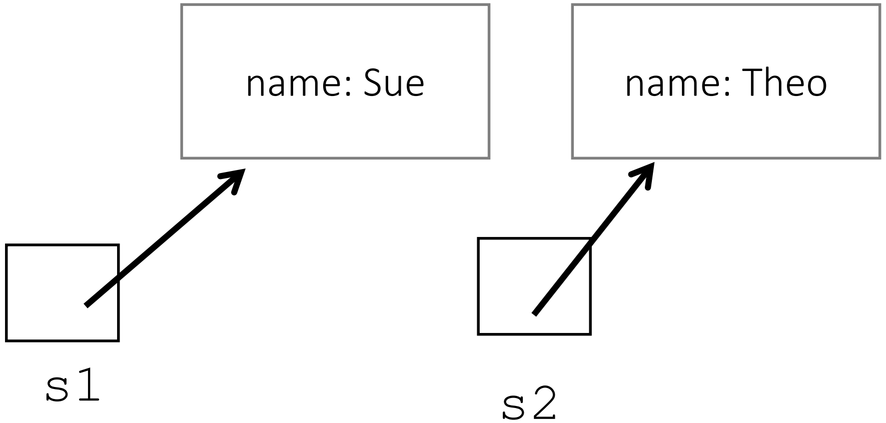
98

num2

Assignment- Objects

- Assignment takes the value on the right and stores it in the variable on the left.
  - For objects, the value is a memory address!

```
Student s1 = new Student(“Sue”);
Student s2 = new Student(“Theo”);
```

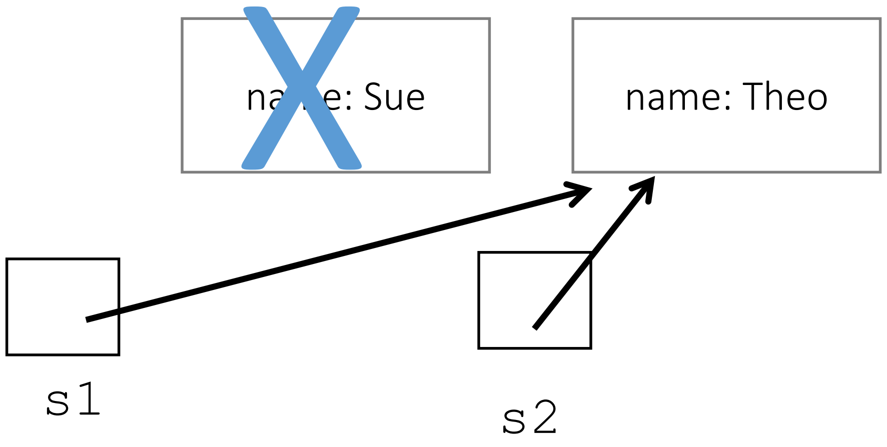


Assignment- Objects

```
s1 = s2;
```

- What is the **value** of s2?
- Because it’s an object, the value is the address!
- So now s1 and s2 point to the exact same place in memory- the same address!

- Because no reference points to the other Student object, it gets garbage collected.

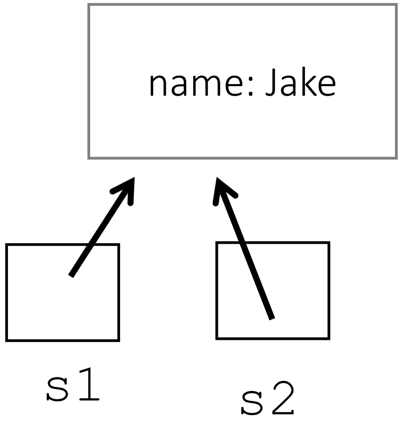


Aliases

- s1 and s2 are now **aliases**.
- Variables that point to the same object (the same place in memory) are *aliases*
- Changing that object through one reference (i.e., one variable name) changes it for *all* references- because there is only **one** object!

Aliases

```
s2.setName(“Jake”);
```



Invoking Methods with Parameters

- Formal parameters are defined in the method header
  - They last as long as the method lasts.
  - When the method is over, these parameters are gone!
- Actual parameters are the values sent when the method is invoked.

Passing Parameters

- When a method is invoked, it’s as if there is assignment statement executed behind the scenes:

```
formalParam = actualParam;
```

- This is an assignment statement!
    - When you use the assignment operator with objects, you create **aliases**.
    - Formal object parameters are **aliases** of actual parameters.

Pass By Value

- Parameters in Java are ***passed by value***
- This means that the ***value*** of the actual parameter is *assigned to* the formal parameter.
  - But remember how assignment works for primitives vs objects!

Objects as Parameters

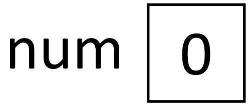
- When an object is passed to a method, the actual parameter and the formal parameter become *aliases* of each other
    - If you change the internal state of the formal parameter by invoking a method, you change it for the actual parameter as well

Review the PassingParameterExample

- In this example, we pass a primitive and an object into a method.

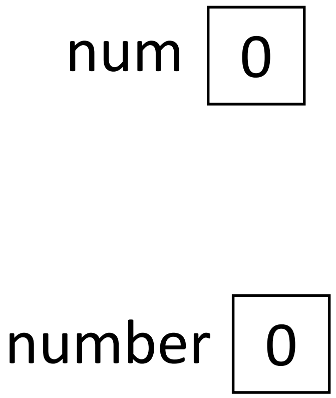
Code Trace- Primitives

```
int num = 0;
```



Code Trace- Primitives

```
primitiveParam(num);
// number = num
// value is assigned!
```

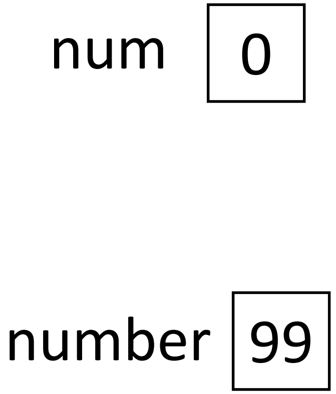


num 0

number 0

Code Trace- Primitives

```
number = 99;
```

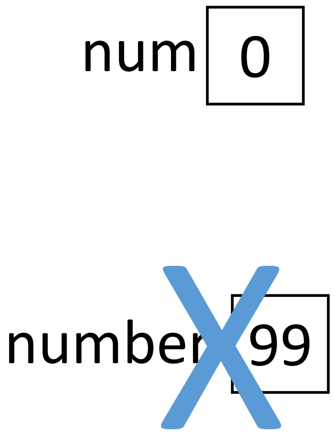


num 0

number 99

Code Trace- Primitives

```
// method ends
// local variables and
   formal parameters
   are garbage collected
```

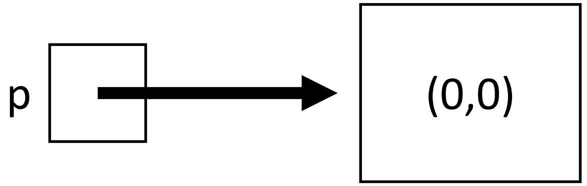


num 0

number 99

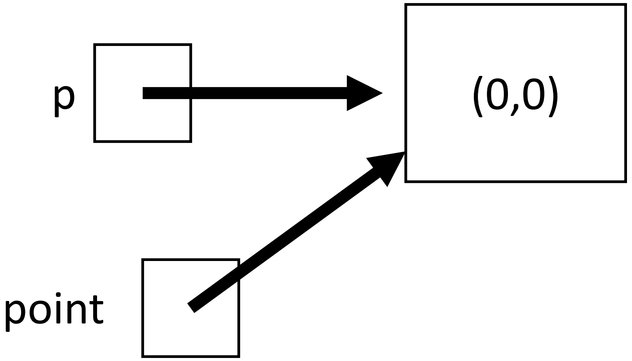
Code Trace- Objects

```
Point p = new Point(0,0);
```



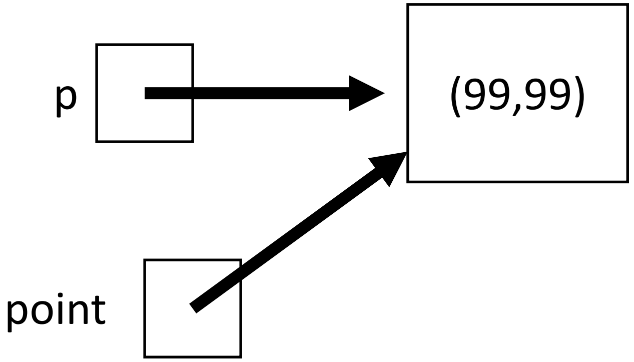
Code Trace- Objects

```
objectParam(p);
// point = p
// value is assigned!
// alias is created!
```



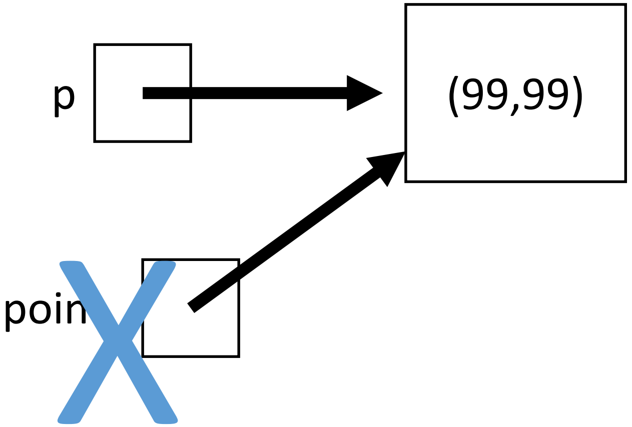
Code Trace- Objects

```
point.setLocation(99,99);
```



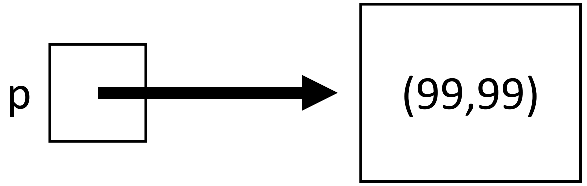
Code Trace- Objects

```
// method ends
// local variables and
   formal parameters
   are garbage collected
```



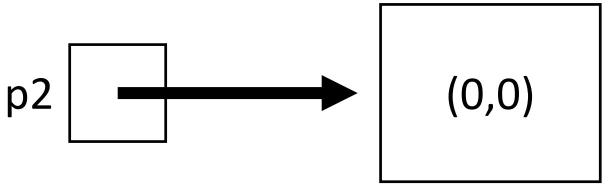
Code Trace- Objects

```
// value is still changed back in main
```



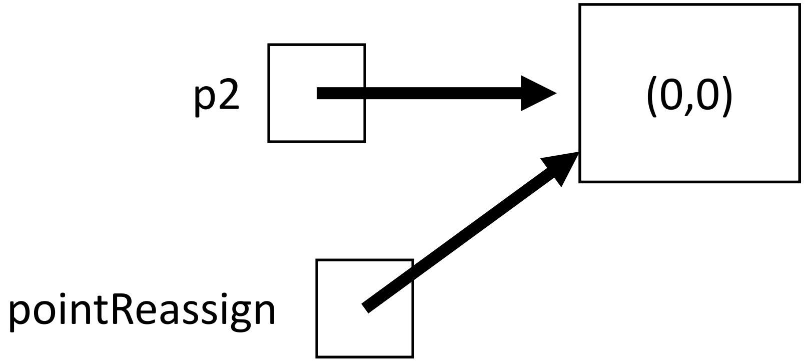
Code Trace- Objects

```
Point p2 = new Point(0,0);
```



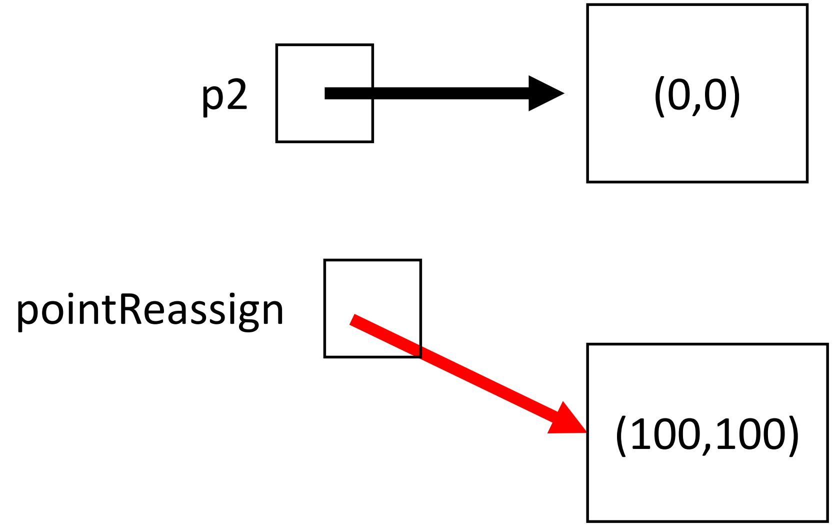
Code Trace- Objects

```
objectParamReassign(p2);
// pointReassign = p2
// value is assigned!
// alias is created!
```



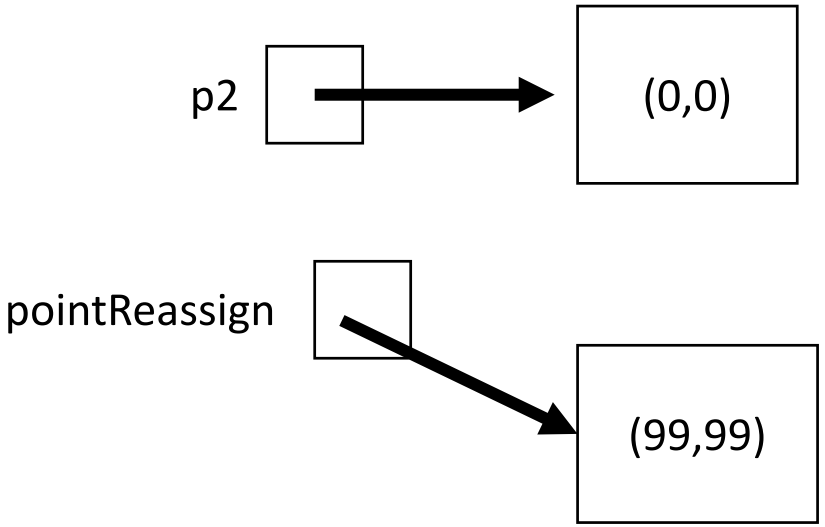
Code Trace- Objects

```
pointReassign = new Point(100,100);
// alias is broken!
```



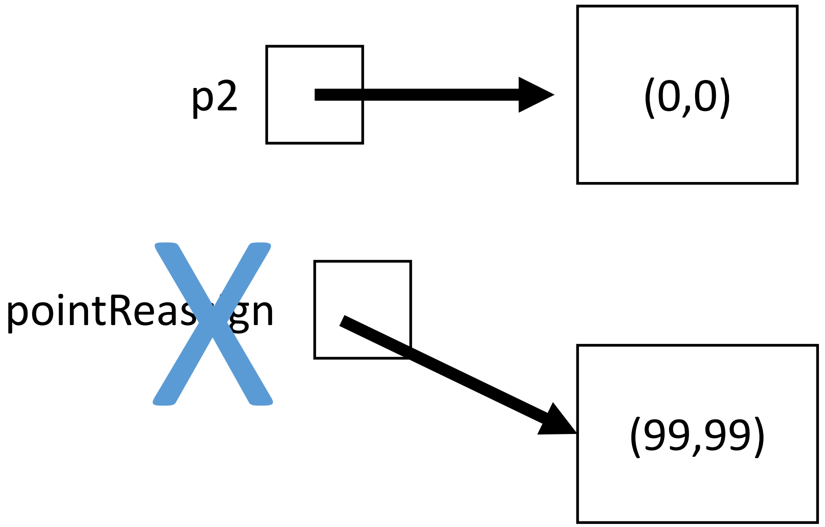
Code Trace- Objects

```
pointReassign.setLocation(99,99);
```



Code Trace- Objects

```
// method ends
// local variables and formal parameters
are garbage collected
```



Key Points about Pass By Value

- Java is pass by value!
- The key is: what is the value?
    - For primitives: the actual data
    - For objects: the memory location/reference
- Using direct assignment with objects creates aliases. (**NOT** copies!)
- Passing objects to parameters creates aliases.
    - Invoking a method (with dot operator) inside the method changes the object inside and outside the method- because it's the same object!
    - Reassigning a formal parameter (formal parameter on left side of equal sign) breaks the alias link. This is usually a mistake.

static

Instance Data Variables Revisited

- Instance data variables
  - Declared in the class
  - Used anywhere in the class
  - Lives as long as the object lives
  - One version for each object

Instance Variables

- For instance variables, each object has its own data space

  ```
  private String firstName;
  ```

  - Each Student has its own first name.
- You update instance data through public methods invoked on an object.

  ```
  student1.setFirstName(“Jim”);
  ```

  - Change the `firstName` of the object `student1`

Static Variables

- Static variables (also called class variables) are associated with the class itself, not with any single instance of the class
- One copy/version for the whole class!

Static Variables

- For static variables, only **one** copy of the variable exists for *all* objects of that class

```
private static int numberOfStudents;
```

  - There is only one count of the number of students and it is shared by all objects of the Student class.
  - If `student1` updates `numberOfStudents`, it’s changed for `student2` as well, because it’s the *same variable!*

Static Variables

- You reference static variables through the name of the class, not through any particular object.
  - `Student.numberOfStudents;`
  - `ButtonType.YES`
  - `Integer.MAX_VALUE`

Static Variables

- Changing the value of a static changes it for all objects of that class

```
public Student(…)     {
  …
  Student.numberOfStudents++;
}
```

- Memory space for static variables is created when the class is first referenced.

Invoking Methods Revisited

- Most methods are invoked through an instance of a class:
  - We create an instance with the new operator
  - We invoke a method with the dot operator
- Examples:

  -
    ```
    Scanner scan = new Scanner(System.in);
    scan.nextLine()
    ```

  -
    ```
    Employee e1 = new Employee(“Ed”);
    e1.pay();
    ```

Static Methods

- Static methods (also called class methods) are invoked **not** through an object but through the **class** name
  - `double answer = Math.sqrt(25)`
  - `double number = Math.random();`
- Static methods are more like *functions* associated with the class
  - They should not be used if a method represents an object’s functionality.
  - They **cannot** be used if they require access to instance data variables.

Static Methods

- Static methods are invoked through the class, not through any object.

```
private static int numberOfStudents;

public static int getNumberOfStudents() {
        return numberOfStudents;
}
public static void setNumberOfStudents(int n){
        numberOfStudents = n;
}
```

Static Methods and Variables

- We declare static methods and variables with the `static` keyword
- A static method or variable is associated with the *class itself*, rather than with any individual instantiated object of the class
  - One copy/version for the whole class!
- By convention, visibility modifiers come first
  - `public static` (not `static public`)

Static Methods and Variables

- Static methods:
    - *cannot* reference instance variables
        - Those variables don’t exist until an object exists
        - And each object has its own version of them
    - *can* reference static variables and local variables

- Static methods:
    - *cannot* directly reference other non-static methods
        - Those must be referenced through an object
    - *can* reference other static methods

- You will get a compiler error if you try to do these things!

Accessing Variables and Methods

	Static Variables	**Instance Variables**
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

```
public Student () {
    …
    Student.numStudents++;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

```
public String getFirstName() {
   return firstName;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

```
public static int getNumStudents() {
   return Student.numStudents;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	**can access**	**cannot access**
Instance Methods	**can access**	**can access**

```
public static int getStudentName() {
   return firstName;
}
```

would be invoked:

```
Student.getStudentName();
```

static methods are invoked through the class, not through an object… so which student's name should be returned??

Using Static Variables

- Shared data (be careful!)
    - Example: a count of objects
    - Example: a total across all objects
- Shared constants
    - Example: `MAX_VALUE`
    - Example: `Math.PI`
    - Example: `Integer.MAX_VALUE`

Using Static Methods

- Utility or helper functions
  - send input, get a result
  - Example: Math.sqrt
- Accessing static variables or shared information
  - Example: getNumberOfstudents()

Practice

- Modify the Employee class to keep track of how many employees have been created
- Modify the PartTimeEmployee class to keep track of the total number of hours worked by all current part time employees

Interfaces

Interfaces (Java 7 and below)

- A Java *interface* is a collection of abstract methods and constants
  - An abstract method can be declared with the modifier `abstract`
- As of Java 8, interfaces can now also contain *default methods*, which are implemented.

Interfaces

- An interface is used to establish a set of methods that a class will implement
  - It's like a contract
- An interface is declared with the reserved word `interface`
- A class indicates that it is implementing an interface with the reserved word `implements` in the class header

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable {
       void doThis();
       int doThat(int num);
}
```

Often public and abstract are left off since these are the defaults.

A semicolon immediately follows each method header

Interfaces

```
public class CanDo implements Doable{
  public void doThis ()
  {
    // whatever
  }

  public void doThat (int num)
  {
    // whatever
  }

  // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

Interface Constants

- Interfaces can also provide public, final, static constants.

Properties of Interfaces

- An interface cannot be instantiated
- Methods of an interface have public visibility
- If a parent class implements an interface, then by definition, all child classes do as well.
  - That functionality is inherited!

Properties of Classes that Implement an Interface

- Provide implementations for *every* method in the interface
    - Can choose whether to override default methods.
- Can have additional methods as well
- Have access to the constants in that interface
- Can implement multiple interfaces but must implement all methods in each interface

```
class DoesALot implements interface1, interface2 {
   // all methods of both interfaces
}
```

Abstraction

- Hiding the details of implementation
- The client only knows about the functionality- what the object does, not how it does it
- Supported through abstract classes and interfaces

Using Interfaces

- It is a design decision whether or not to have a class implement an interface.
- Often, interfaces describe common *functionality* across classes rather than common *features* (which is more suited for inheritance)
  - Inheritance “is a”
  - Interface “does …” “can …” “is …able”
- Interfaces are Java’s way of ensuring that a class contains an implementation for a specific method.
  - That an object has a specific functionality.

Interfaces and Polymorphism

- An interface can be used as a declared type.
  - But not as an actual type, since you cannot instantiate it!
- The variable can be instantiated with any class that implements the interface
  - The method that is invoked is based on the actual type.
  -

Interfaces and Polymorphism

```
public interface Speaker {
   public abstract void speak();

}

public class Dog extends Animal implements Speaker {

      public void speak() {

              System.out.println(“Woof”);

      }

}

public class Parrot extends Bird implements Speaker {

      public void speak() {

              System.out.println(“Polly wants a cracker…”);

      }

}
```

```
Speaker[] speakers = new Speaker[2];

speakers[0] = new Parrot();

speakers[1] = new Dog();

for(Speaker sp : speakers) {

      sp.speak();

}
```

The Comparable Interface

The `Comparable` Interface

- Specifies that two objects can be *compared* or *ordered* with each other.
- The `compareTo` method defines how that ordering is done.
- Many Java classes implement compareTo.
  - `String`, which is why we can call the `compareTo` method on two `Strings`
- Any class we write can implement `Comparable`
  - We decide how our objects are ordered.

Comparing Objects

- Sorting items is a common thing to do. There are many different ways to sort objects.
- All methods of sorting, however, at some point involve comparing two objects to each other- is object A less than, greater than, or equal to object B?
- Implementing the Comparable interface allows us to provide a method for how to make this comparison.
- This is called the *natural ordering* of objects.

The `Comparable` Interface

- The Java standard class library contains the `Comparable` interface which has one abstract method used to compare two objects

```
    public int compareTo(Object obj)
```

- Use generics to improve the method:

```
     public MyClass implements
                        Comparable<MyClass>
          public int compareTo(MyClass obj)
```

The `Comparable` Interface

- The value returned from `compareTo` is:
  - negative if `obj1` is less than `obj2`
  - 0 if they are equal
  - positive if `obj1` is greater than `obj2`

```
if (obj1.compareTo(obj2) < 0)
   // obj1 less than obj2
else if(obj1.compareTo(obj2) > 0
  // obj1 greater than obj2
else
  // they are equal
```

The `Comparable` Interface

- It’s up to you how to determine what makes one object greater to, less than, or equal to another
  - Example: For an `Employee` class, you could order employees by name (alphabetically), by employee ID number, or by start date
- The implementation of the `compareTo` method can be as straightforward or as complex as needed

The `Comparable` Interface

- Implementing the `Comparable` interface allows us to use nice methods from the Java standard class library, such as sorting methods.
  - Collections.sort(myArrayList)
  - Arrays.sort(myArray)
- These methods only works if the class implements `Comparable`

Comparable and Sorting

- Note that implementing compareTo doesn’t actually sort anything!
- It only defines *how* to compare two objects to each other.
- This is needed in order to sort. But to actually do the sort, we need another method.

Practice

- Implement the Comparable interface in the Employee class.
- Sort a list of employees.

Enumerated Data Types (enums)

enum

- A way to provide a restricted set of values
- You can declare variables of this type.
- Examples
  - Sizes: Small, medium, large
  - Suits: Diamonds, hearts, spades, clubs
  - Semesters: Fall, summer, spring

```
enum Size {SMALL, MEDIUM, LARGE};

Size s = Size.LARGE;
// s can only hold the values SMALL, MEDIUM, LARGE, null
```

Can’t we just use constants?

```
public static final int SMALL = 0;
public static final int MEDIUM = 1;
public static final int LARGE = 2;

public int size = SMALL;
```

- No type safety
  - `public void setSize(int size) {` // someone could send in -9!
- Allows for illogical results
  - If you had `public static final int FALL = 2;` then `FALL == LARGE`. Huh?!
- Brittle
  - If a constant value is changed, the code must be recompiled, and so must any other code that uses that value.
- No easy way to translate to String output
- No way to iterate over all of the choices

Constants vs. enums

- Bottom line: Constants are good things! You should use them in your code. But enums are better when they make sense.
- Constants are good for single values like min, max, default values, etc.
- enums are good when something has a predefined, finite set of possible values.

Practice

- Add an enums to our classes to represent a *status* that describes all employees.
- Use that enum in the class- only active, full time employees receive benefits.

enums are actually classes!

- An enum is actually a class with exactly X number of instances declared. And it’s not possible to construct any more instances.
- Can add constructors, methods, and fields.
    - Constructors are invoked when the enum constants are constructed.
    - Methods and fields are used when you want to associate data or behavior with a constant
- All enums are subclasses of `Enum`.
- You can use == to compare enum types.

Example (from Core I)

```
enum Size {
      SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

      private String abbreviation;

      private Size(String abbreviation) {
            this.abbreviation = abbreviation;
      }
      public String getAbbreviation() {
            return abbreviation;
      }
}
```

The Enum Class- Inherited Methods

- `toString`
  - returns the name of the constant
  - invoke on an enum’s value
  - example: Size.SMALL.toString() returns “SMALL”
- `valueOf`
  - send in the class and the name and get back the value
  - static method invoked on Enum class
  - example: Size s = Enum.valueOf(Size.class, “SMALL”);

The Enum Class- Inherited Methods

- `values`
  - returns an array of all possible values
  - static method invoked on the actual enum
  - example: Size[] values = Size.values();
- `ordinal`
  - returns the position of the constant in the declaration (starting from 0)
  - invoke on an enum’s value
  - example: Size.LARGE.ordinal() returns 2

enums- Better than Constants!

- Type safety
  - `public void setSize(Size s) {`

    // can only accept a variable of type Size
- No illogical results
  - Size.SMALL is not equal to Semester.FALL
- Flexible
  - You can add onto your enum values- in any order- and other code that uses the enum will continue to work. No recompiling needed!
- Easy String output through enum methods.
- Easy iteration over all possible choices through values() method.

Practice

- Revise the Status enum to add more information using a constructor and method.

Practice

- Write an enum to represent ice cream flavors.
  - Some flavors are nut-free, others are not.
- Write a class to represent an ice cream order (described by flavor and number of scoops).
- Write a driver program to ask the user to create orders by entering the flavors (taking nuts into consideration!) and number of scoops.

Practice

- Write code related to a CustomerAgent (perhaps who works on the phone doing customer service) and a FeedbackReport that is made about an agent (perhaps by the customer after he/she is helped).
- Write an enum to describe the range of the score (negative, neutral, positive).
    - Include variables for the range of scores.
    - Add a method to return an enum value based on the score.
- Write a FeedbackReport class:
    - Described by the score, range, and text feedback
    - Each report has a unique ID generated when the report is created
    - Keep track of the number of all negative feedback reports created
- Write a CustomerAgent class:
    - Described by name and a list of FeedbackReports
    - Write a method to determine if an agent is eligible for a bonus- they are eligible if they have had at least 100 feedback reports and 75% or more of them are positive
    - Write a method to clear out an agent's feedback report (perhaps at the start of a new year)

On Your Own Practice

- Use the Store Inventory classes from Module 01.

- Add a static variable and static method.
- Implement the Comparable interface in one of your classes.
  - In your tester program, sort a list of objects.
- Add at least one enum with at least one field and method. Use this enum somewhere in the classes.