Class Design Part 2

null

null

- null is a keyword/reserved word in Java
- null represents **no value**

null as a Default Value

- Instance data variables are given a default value when they are declared.
  - Local variables are not.
- Each data type has a value that is used for the default value.
- null is the default value for objects when they have been declared but not initialized.
  - private int age; // default value for int is 0
  - private String name; // default value of name is null
  - private Student s; // default value of s is null

null as a Value

- null is a value that can be assigned to any *object* reference (variable).
- null cannot be assigned to primitives.
- Examples:
  - Student s = null; // allowed
  - int n = null; // not allowed
  - Integer m = null; // allowed

What can you do with null?

- Compare it with ==
- Examples:
  - if(student!=null) { … }
  - if(student!=null && student.meetsCriteria()) { … }

What **can't** you do with null?

- Pretty much anything else!
- Most importantly: you cannot invoke any methods or try to access any variables on null.
- Invoking a method on a null object will throw a NullPointerException, which will crash your program.
  - This is a bad exception because it is almost always a result of programmer error.
  - This is almost always entirely preventable.

What **can't** you do with null?

- Example:
  - String s = null
  - s.equals("hello"); // crash!
  - s=="hello"; // allowed- will not crash, returns false
- Example:
  - Student student = null;
  - System.out.println(student.name); // crash!

null and Strings

- A string whose value is the empty String is **not** null.
- An empty String is a String that does not contain any characters. (But it is not null!)
  - String s1 = “”; // empty string! length = 0
  - String s2 = null; // no length- no value!
  - s1==s2; // false

null and Strings

```
String s1 = "";
String s2 = null;
String s3; // value is also null

System.out.println(s1.toUpperCase());
// allowed- but nothing will be printed!

System.out.println(s1.length());
// allowed and will print 0

System.out.println(s2.toUpperCase());
// not allowed- will crash with NullPointerException

System.out.println(s3.length());
// not allowed- will crash with NullPointerException
```

null Exceptions

- Provides the null variable and other helpful information with NullPointerExceptions
- If you are running older versions, you might need to enable this in your VM arguments:

```
-XX:+ShowCodeDetailsInExceptionMessages
```

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!

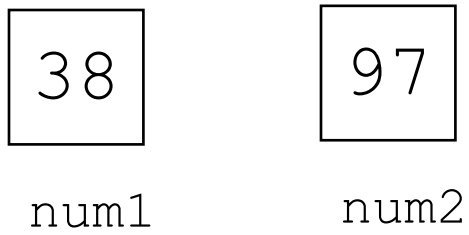
Assignment Statements

- 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!

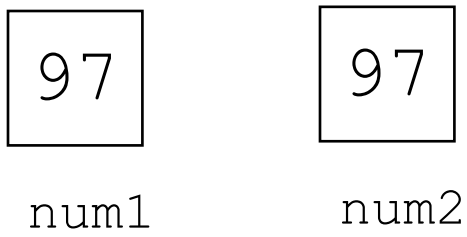
```
int num1 = 38;

int num2 = 97;
```



Assignment- Primitives

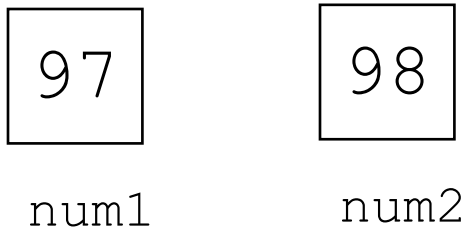
```
num1 = num2;
```

- 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.



Assignment- Primitives

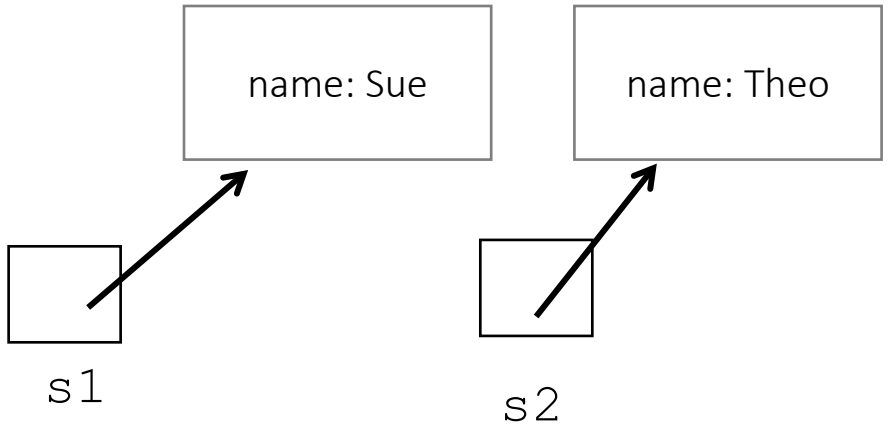
```
num2++;
```



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");
```

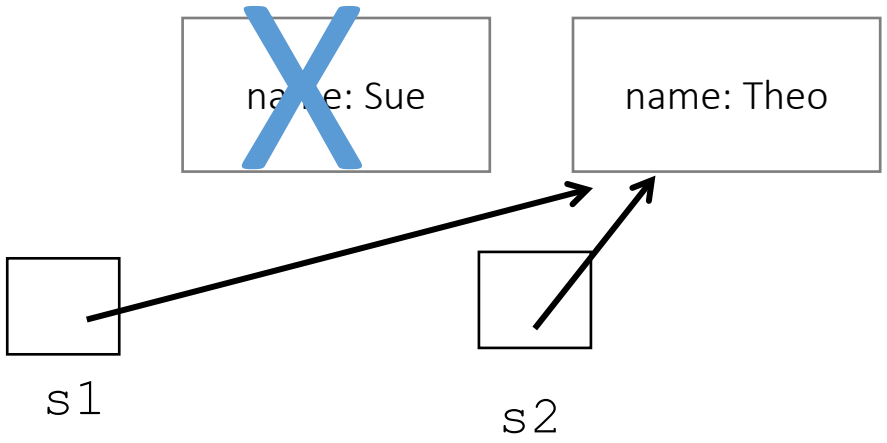


Assignment- Objects

```
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.

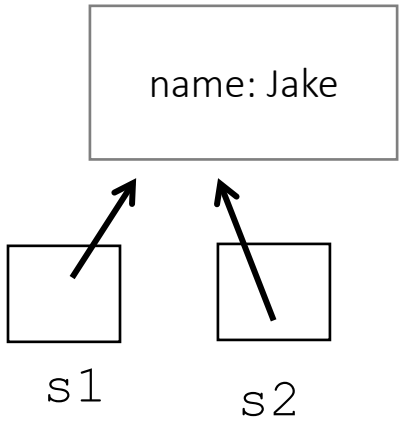


Aliases

- 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

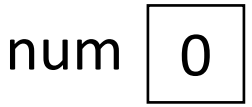
- When an object is passed to a method, the actual parameter and the formal parameter become *aliases* of each other
  - If you change the internal state of the formal parameter by invoking a method, you change it for the actual parameter as well

Review the PassingParameterExample

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

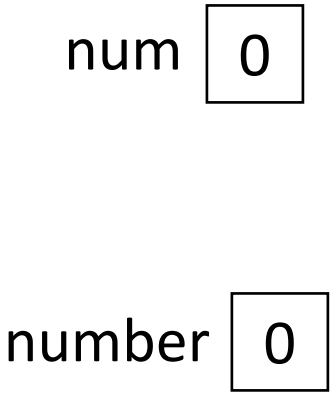
Code Trace- Primitives

```
int num = 0;
```



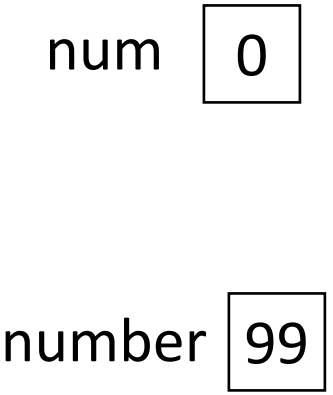
Code Trace- Primitives

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



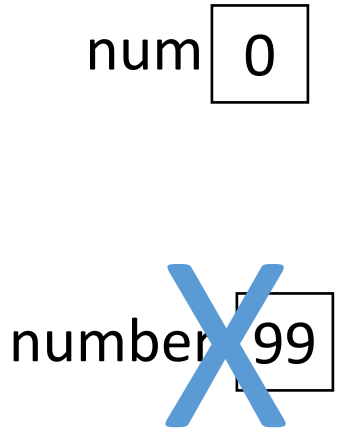
Code Trace- Primitives

```
number = 99;
```



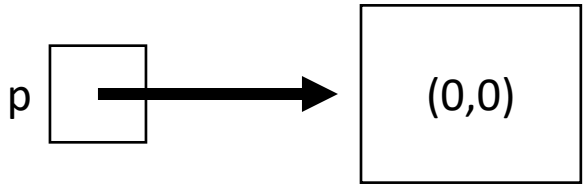
Code Trace- Primitives

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



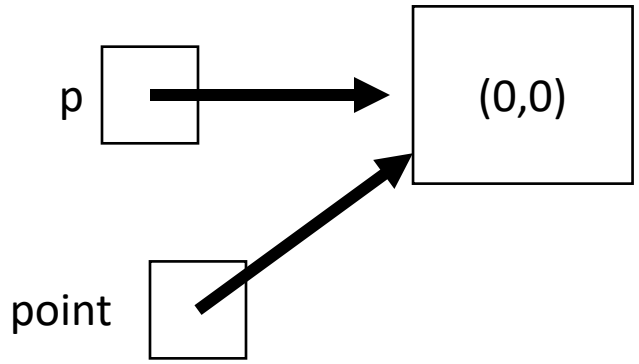
Code Trace- Objects

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



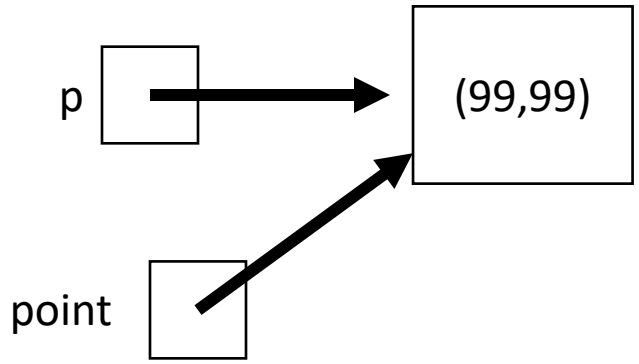
Code Trace- Objects

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



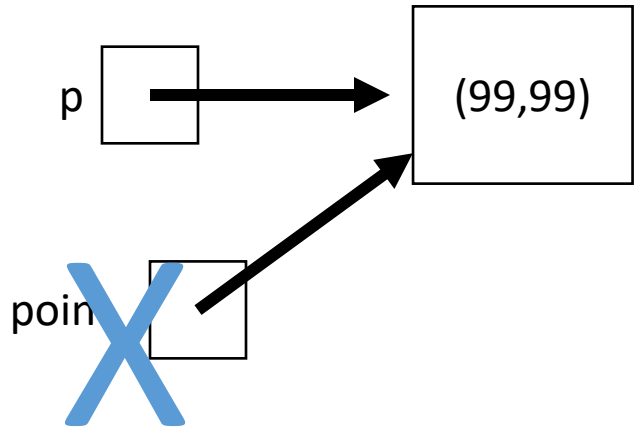
Code Trace- Objects

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



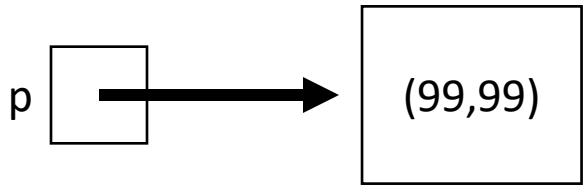
Code Trace- Objects

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



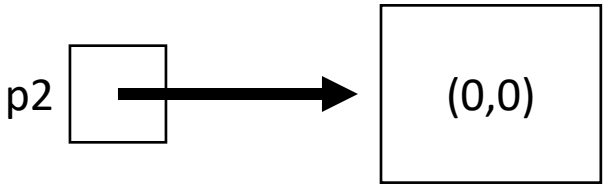
Code Trace- Objects

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



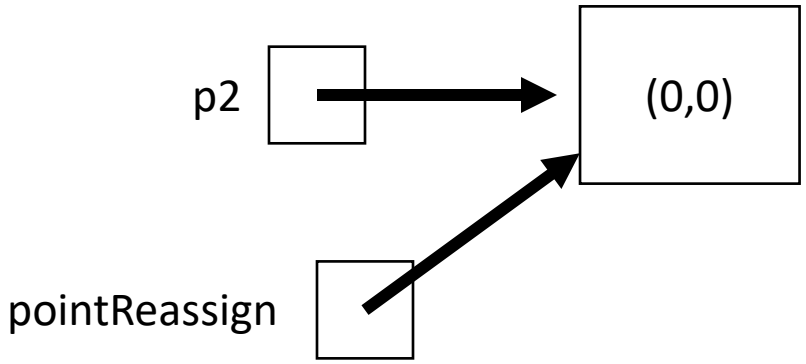
Code Trace- Objects

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



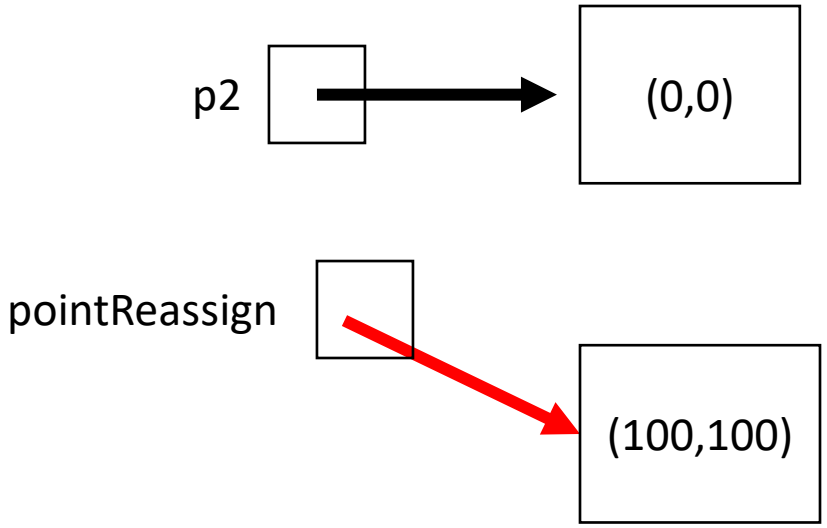
Code Trace- Objects

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



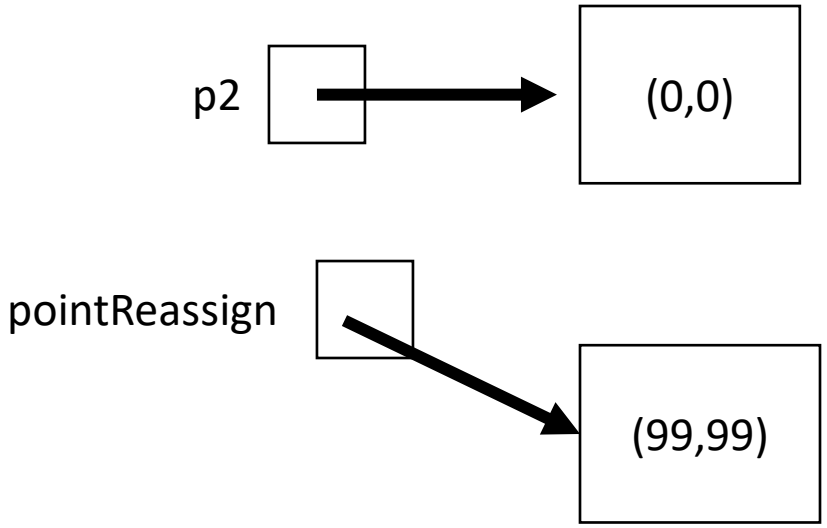
Code Trace- Objects

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



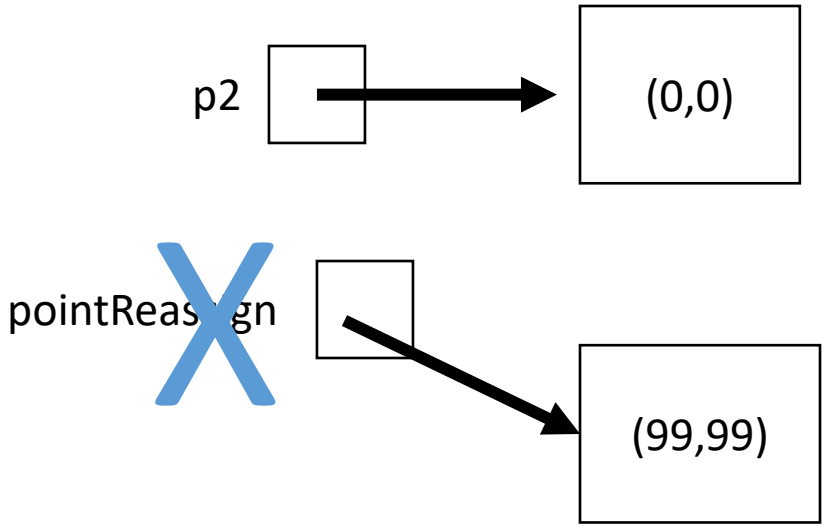
Code Trace- Objects

```
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

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

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

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.

Records

New to Java 16: Records

- A Java Record is a special class that provides a concise way to define an immutable "data-only" class.

A Data-Only Class

- final instance data variables
- constructor that takes in parameters and initializes variables
- getters
- toString
- equals
- hashCode

```java
import java.util.Objects;

public class StudentClass {

    private final String name;
    private final int id;

    public StudentClass(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
        return "StudentClass [name=" + name + ", id=" + id + "]";
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        StudentClass other = (StudentClass) obj;
        return id == other.id && Objects.equals(name, other.name);
    }

}
```

A Record

```
public record StudentRecord (String name, int id ) { }
```

Creating a Record

- Use the keyword record instead of class
- List the instance data variables in parentheses

```
public record StudentRecord (String name, int id ) { }
```

Records- Behind the Scenes

```
public record StudentRecord (String name, int id ) { }
```

- Java automatically generates:
  - private, final instance data variables
  - a *canonical* constructor with parameters for each variable
  - getter methods (named the same as the variable x- **not** getX)
  - an equals method that defines logical equivalence as all variables being equal
  - a hashCode method
  - a toString method that includes the name of the class and the name and value of each variable

Benefits of Records

- Reduces the need for boilerplate code.
- Makes the purpose of the class clearer.
- Makes the class more maintainable if we add variables.
- (Side bonus for us: much less tedious!)

Using Records

- You use records the same way you would use any ordinary class because a record **is** a class.
  - It's just a special kind of class- just like an enum is a special kind of class.

```
2  public class RecordTester {
3
4      public static void main(String[] args) {
5          StudentClass studentC = new StudentClass("Jessica", 123);
6          System.out.println(studentC);
7          System.out.println(studentC.getName());
8
9          StudentRecord studentR = new StudentRecord("Cedric", 567);
10         System.out.println(studentR);
11         System.out.println(studentR.name());
12     }
13
14 }
15
```

Problems Javadoc Declaration Console ×

```
<terminated> RecordTester [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe  (Nov 17, 2022, 11:12:49 AM – 11:12:49 AM) [pid: 49344]
StudentClass [name=Jessica, id=123]
Jessica
StudentRecord[name=Cedric, id=567]
Cedric
```

Adding More to a Record

- You can add additional constructors.
  - Each additional constructor **must** invoke the behind-the-scenes constructor using this(…).
- You can add instance methods.
- You can add static variables/constants and methods.

- Note that you can also explicitly declare any of the behind-the-scenes methods if you want to modify them.

Modifying the Behind-the-Scenes Constructor

- We can add code to the constructor that is generated for us.
  - This should primarily be used for validating or altering parameters.
- We can create a *compact constructor* that specifies code that should be added to the canonical constructor **before** the variables are initialized.

```
public record StudentRecord (String name, int id ) {

    public StudentRecord {
        name = name.toUpperCase();
    }

}
```

```
public record StudentRecord (String name, int id ) {

    public StudentRecord {
        if(id<0) {
            System.out.println("Error with student id: " + id);
        }
    }

}
```

Records and Inheritance

- All Records extend the class Record.
  - Your Record cannot have another parent.
- All Records are final.
  - You cannot extend your Record.
- Records **can** implement an interface.

- Also note: Records **can** be generic.
  - Example: `public record Pair<T> (T item1, T item2) { }`

FACTORIES

Creating Objects

- Review the simplified Employee classes.
- Write a Department class that creates Employee objects and adds them to a list for that department. The department also has a method that conducts a review of an employee.

Is our solution flexible?

- What if we suddenly have interns in addition to full time and part time employees?
- We'd have to update the add method in the Department class…
  - And the HRDepartment class…
  - And also **all other classes** where Employee objects are created. That could be a lot of places!

Is our solution flexible?

- No!
- It would be better if we could separate the object *creation* from the object *processing*.
- The processing will always stay the same: we want to add an employee to the list, or we want to conduct a review of the employee.
- But the way we create an Employee object might change- it's conceivable that we might add more kinds of Employees.
- Separate what varies from what stays the same.

Factory Classes and Methods

- A factory method creates objects.
  - Usually static, but not always.

- A factory class contains a collection of factory methods.

- You might have used factory methods:
  - NumberFormat.getCurrencyInstance()
  - NumberFormat.getPercentInstance()
  - Calendar.getInstance()
  - myArrayList.iterator()

Static Factory Methods

- You have a parent/base class (usually abstract) (or could also be an interface).
- You have several child/sub classes (or several classes that implement the interface).
- The factory method returns the parent/base class.
  - Factory method decides which child class to instantiate.
- Common names
  - valueOf
  - of
  - getIntance
  - newInstance
  - getType
  - newType

Practice

- Write a simple static factory method class to create Employee objects.
- Add an Intern class and

Is our solution flexible?

- Much better! We've separated the creation of the object from the processing of the object.
  - Department has no knowledge of what kind of Employee it is processing. It doesn't need to know!
  - In the future, we can update just the factory method if we want to add new subclasses.

Benefits of Factories

- Can choose from multiple child classes and return a subtype
  - The client only needs to know about the parent class functionality- it doesn’t need to know the full class hierarchy
- Can reuse objects (e.g., database connections)
- Can return null
- Can have descriptive names and can support different interpretations of the same parameter types
  - Student.newInstanceByName(String name)
  - Student.newInstanceByID(String id)
- Limitations
  - Classes with private constructors cannot be extended.
  - Factory methods are not always easily identifiable.

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 one or more static factory methods.
  - Consider what “type” characteristics you want to use to create your items.
  - Consider whether you want to use a method or class.
  - In your tester program, create objects using your factory.
- Write a record.
  - Consider adding an additional constructor or method or creating a compact constructor.