11/18/22

Replit Notes: MusicCollection

```
private void showSongsBy(String artist) {
  System.out.println("Showing songs by " + artist + ":");
  if (!filterMusic(artist, null, SONGS_MODE)) {
    System.out.println("No songs by that artist found.");
  }
}

private void showAlbumsBy(String artist) {
  if (!filterMusic(artist, null, ALBUMS_MODE)) {
    System.out.println("No albums by that artist found.");
  }
}

private void showSongs() {
  if (!filterMusic(null, null, SONGS_MODE)) {
    System.out.println("No songs found! Load some music up!");
  }
}

private void showAlbums() {
  if (!filterMusic(null, null, ALBUMS_MODE)) {
    System.out.println("No albums found! Load some music up!");
  }
}

private void showArtists() {
  if (!filterMusic(null, null, ARTISTS_MODE)) {
    System.out.println("No artists found! Load some music up!");
  }
}

private void showSongsOn(String album) {
  if (!filterMusic(null, album, SONGS_MODE)) {
    System.out.println("Could not find that album.");
  }
}
```

```
private boolean filterMusic(String artist, String album, int mode) {
  MusicScanner musicScanner = new MusicScanner("songs.txt");
  HashSet<String> alreadySeen = new HashSet<String>();
  boolean foundAny = false;
  Song song;
  while ((song = musicScanner.next()) != null) {
    if (artist != null && !artist.equalsIgnoreCase(song.getArtist())) {
      continue;
    }
    if (album != null && !album.equalsIgnoreCase(song.getAlbum())) {
      continue;
    }
    if (mode == ALBUMS_MODE) {
      if (alreadySeen.add(song.getAlbum())) {
        if (artist != null) {
          System.out.println(song.getAlbum());
        } else {
          System.out.println(song.getAlbum() + " (by " + song.getArtist() + ")");
        }
      }
    } else if (mode == ARTISTS_MODE) {
      if (alreadySeen.add(song.getArtist())) {
        System.out.println(song.getArtist());
      }
    } else {
      System.out.println(song);
    }
    foundAny = true;
  }
  return foundAny;
}
```

Sections 5.7, 5.8, 5.9

Statics
Scope and Access
`this`

Statics

- Static variables & methods belong to Classes - not Instances of a Class
  - There is only one copy of a `static` variable & method
  - They can be `public` or `private`
- Static variables & methods are accessed using the name of the class to which they belong and a dot (`.`) With a couple of exceptions: Statics accessing statics; And static imports
  - `Math.PI`
  - `Math.random()`
  - `Math.sqrt()`
- The `main()` method is a static method - It is only ever run one time for a program - And the JVM needs to run it without creating an Instance of a Class

Statics

- Static variables & methods are accessed using the name of the class to which they belong and a dot (.)

Exception A static can invoke another static without the class name prefix

```
class Population {
  public static int getNumAdults() {
    return numAdults;
  }
  public static int getNumChildren() {
    return numChildren;
  }
  public static int getPopulation() {
    return getNumAdults() + getNumChildren();
  }
}
```

```
Population.getNumAdults();

Population.getNumChildren();

Population.getPopulation();
```

Statics

- Static variables & methods are accessed using the name of the class to which they belong and a dot (`.`)

Exception Recall from Section 2.9 that static imports can be used to pull in statics from another package to your default scope - which you can then invoke without the Class name

```
import static java.lang.Math.*;


public class Main {
  public static void main(String args[]) {
    System.out.println(PI);
    System.out.println(sqrt(9));
  }
}
```

Statics

- Statics can directly access other Statics
- Statics cannot directly access non-Statics
- Non-Statics can directly access Statics

Statics

- Statics can directly access other Statics
- Statics cannot directly access non-Statics
- Non-Statics can directly access Statics

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

Statics

- **Statics can directly access other Statics**
- Statics cannot directly access non-Statics
- Non-Statics can directly access Statics

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
Person.getNumPeople();

A> ?
```

Statics

- **Statics can directly access other Statics**
- Statics cannot directly access non-Statics
- Non-Statics can directly access Statics

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
Person.getNumPeople();

A> 0
```

Statics

- Statics can directly access other Statics
- **Statics cannot directly access non-Statics**
- Non-Statics can directly access Statics

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
public static int getNumPeople() {
  // System.out.println(name);
  return numPeople;
}
```

Statics

- Statics can directly access other Statics
- Statics cannot directly access non-Statics
- **Non-Statics can directly access Statics**

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
Person p1 = new Person("Julie");
Person p2 = new Person("Bobby");

p1.report()

B> ?

p1.getNumPeople();

C> ?
```

Statics

- Statics can directly access other Statics
- Statics cannot directly access non-Statics
- **Non-Statics can directly access Statics**

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
Person p1 = new Person("Julie");
Person p2 = new Person("Bobby");

p1.report()

B> Julie is one of 2 people

p1.getNumPeople();

C> 2
```

The Singleton Pattern

Singleton Pattern

- A way to ensure that one AND ONLY ONE Instance of a Class is created
- Used to coordinate data and functionality across components of a software program
- Overall can reduce the need for statics
  - No need to make every method a static when you can ensure that only a single Instance of a Class is ever created
- Examples
  - Generate a unique identifier
  - Isolate and coordinate access to critical shared resources
- Typically built with a static Factory Method and a private Constructor

Wikipedia: Singleton Pattern, Factory Function

Singleton Pattern

```
public class UniqueIdCreator {
 private int uniqueId;
 private UniqueIdCreator() {
   uniqueId = 1;
 }
 public int getUniqueId() {
   return uniqueId++;
 }
 private static UniqueIdCreator instance;
 public static UniqueIdCreator getInstance() {
   if (null == instance) {
     instance = new UniqueIdCreator();
   }
   return instance;
 }
}
```

Singleton Pattern

```
public class UniqueIdCreator {
 private int uniqueId;
 private UniqueIdCreator() {
   uniqueId = 1;
 }
 public int getUniqueId() {
   return uniqueId++;
 }
 private static UniqueIdCreator instance;
 public static UniqueIdCreator getInstance() {
   if (null == instance) {
     instance = new UniqueIdCreator();
   }
   return instance;
 }
}
```

Singleton Pattern

```
public class UniqueIdCreator {
 private int uniqueId;
 private UniqueIdCreator() {
   uniqueId = 1;
 }
 public int getUniqueId() {
   return uniqueId++;
 }
 private static UniqueIdCreator instance;
 public static UniqueIdCreator getInstance() {
   if (null == instance) {
     instance = new UniqueIdCreator();
   }
   return instance;
 }
}
```

Singleton Pattern

```
public class UniqueIdCreator {
 private int uniqueId;
 private UniqueIdCreator() {
   uniqueId = 1;
 }
 public int getUniqueId() {
   return uniqueId++;
 }
 private static UniqueIdCreator instance;
 public static UniqueIdCreator getInstance() {
   if (null == instance) {
     instance = new UniqueIdCreator();
   }
   return instance;
 }
}
```

Singleton Pattern

```
public class UniqueIdCreator {
 private int uniqueId;
 private UniqueIdCreator() {
   uniqueId = 1;
 }
 public int getUniqueId() {
   return uniqueId++;
 }
 private static UniqueIdCreator instance;
 public static UniqueIdCreator getInstance() {
   if (null == instance) {
     instance = new UniqueIdCreator();
   }
   return instance;
 }
}
```

```
UniqueIdCreator uic = new UniqueIdCreator();
```

Singleton Pattern

```
public class UniqueIdCreator {
 private int uniqueId;
 private UniqueIdCreator() {
   uniqueId = 1;
 }
 public int getUniqueId() {
   return uniqueId++;
 }
 private static UniqueIdCreator instance;
 public static UniqueIdCreator getInstance() {
   if (null == instance) {
     instance = new UniqueIdCreator();
   }
   return instance;
 }
}
```

```
UniqueIdCreator uic = new UniqueIdCreator();

** ERROR ** CONSTRUCTOR IS PRIVATE - CANNOT USE NEW
```

Singleton Pattern

```
public class UniqueIdCreator {
 private int uniqueId;
 private UniqueIdCreator() {
   uniqueId = 1;
 }
 public int getUniqueId() {
   return uniqueId++;
 }
 private static UniqueIdCreator instance;
 public static UniqueIdCreator getInstance() {
   if (null == instance) {
     instance = new UniqueIdCreator();
   }
   return instance;
 }
}
```

```
UniqueIdCreator uic = new UniqueIdCreator();
```

**** ERROR ** CONSTRUCTOR IS PRIVATE - CANNOT USE NEW**

```
UniqueIdCreator.getInstance().getUniqueId();
```

D> ?

```
UniqueIdCreator.getInstance().getUniqueId();
```

E> ?

Singleton Pattern

```
public class UniqueIdCreator {
 private int uniqueId;
 private UniqueIdCreator() {
   uniqueId = 1;
 }
 public int getUniqueId() {
   return uniqueId++;
 }
 private static UniqueIdCreator instance;
 public static UniqueIdCreator getInstance() {
   if (null == instance) {
     instance = new UniqueIdCreator();
   }
   return instance;
 }
}
```

```
UniqueIdCreator uic = new UniqueIdCreator();

** ERROR ** CONSTRUCTOR IS PRIVATE - CANNOT USE NEW


UniqueIdCreator.getInstance().getUniqueId();

D> 1

UniqueIdCreator.getInstance().getUniqueId();

E> 2
```

Scope and Access Control

- Scope of a variable is where a variable can be accessed and used
- Determined by where the variable is declared in the program and can be found by looking at the closest curly brackets
- **Class Level Scope** Instance and static variables inside a Class
- **Method Level Scope** Local variables (including parameter variables) inside a method
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with `{ }`

Scope and Access Control

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```
public class Person {
  private String name;
  private static int numPeople;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

Scope and Access Control

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```
public class Person {
  private String name;
  private static int numPeople;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

Scope and Access Control

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```
public class Person {
  private String name;
  private static int numPeople;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

Scope and Access Control

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```
public class Person {
  private String name;
  private static int numPeople;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

Scope and Access Control

```
public class Person {
  private int age = 10;
  public Person(int age) {
    age = 20;
  }
  public int getAge() {
    return age;
  }
  public void loopTest(int age) {
    int age = 30;
    if (true) {
      int age = 40;
      age = 50;
    }
    age = 60;
  }
}
```

Which variables have

- **Class Level Scope**
- **Method Level Scope**
- **Block Level Scope**

Scope and Access Control

```
public class Person {
  private int age = 10;
  public Person(int age) {
    age = 20;
  }
  public int getAge() {
    return age;
  }
  public void loopTest(int age) {
    age = 30;
    if (true) {
      int age = 40;
      age = 50;
    }
    age = 60;
  }
}
```

A

B

C

D

Which variables have

- **Class Level Scope**
- **Method Level Scope**
- **Block Level Scope**

Which `age` variable is referenced at

- **A**
- **B**
- **C**
- **D**

Scope and Access Control

```
public class Person {
  private int age = 10;
  public Person(int age) {
    age = 20;
  }
  public int getAge() {
    return age;
  }
  public void loopTest(int age) {
    age = 30;
    if (true) {
      int age = 40;
      age = 50;
    }
    age = 60;
  }
}
```

A

B

C

D

EXTREMELY *easy to make an error in your Constructor with parameter and instance variable names.*

We have been using `initAge` *and* `initName` *(as parameter names) to disambiguate between* `age` *and* `name` *instance variables.*

But now we can start using...

d at

`this` (keyword)

- Within an Instance method of a Class - `this` refers to the current Instance (and refers to the Instance being created in a Constructor)
- Static methods cannot refer to `this` (since there is no Instance when using static methods)

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String name) {
    numPeople++;
    this.name = name;
  }
  public static int getNumPeople() {
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

Scope and Access Control

```
public class Person {
  private int age = 10;
  public Person(int age) {
    age = 20;
    this.age = 30;
  }
  public void loopTest(int age) {
    age = 40;
    if (true) {
      int age = 50;
      this.age = 60;
    }
    age = 70;
  }
}
```

A

B

C

D

Which `age` variable is referenced at

- **A**
- **B**
- **C**
- **D**

Reading, Practice, and Assignments

- CSAwesome
  - 5.7. Static Variables and Methods
  - 5.8. Scope and Access
  - 5.9. `this` Keyword
- Monday
  - Unit 5 Project (Hangman) due 11/27 11:59pm
  - Unit 5 Test