11/18/22

Unit 5 Review

5.1
Anatomy of a Java Class

Definitions

Class - Blueprint for an object; instructions for how construct an object. **There can be ONLY ONE of these -> "Dog"**

Object - A particular instance of a class; use the new operator to create an object instance from a Class. **There can be MANY of these -> "A 3 year-old German Shepherd named Roscoe", "A 1 year-old Golden Retriever named Lucy"**

Properties and Methods - **Properties** are attributes (name, age, breed) and **Methods** are operations (play, eat, sleep); and each can be either `public` (available outside the Object) or `private` (available from only the inside of an Object)

Instance Variables

- Also known as attributes, properties, or fields
- Holds the data of an object
- **Every Object instance has their own values for these properties**

```
Dog scout = new Dog("Scout", 10);
Dog bailey = new Dog("Bailey", 5);
```

```
scout.name.equals("Scout") == true
bailey.name.equals("Bailey") == true

scout.name.equals(bailey.name) == false
```

Instance Methods

- Define the behavior generically in the Class
- **So that it can be used by every Object instance**

```
public void feedDog() {
    System.out.println("Gave " + name + " a bowl of food.");
}
```

```
Dog scout = new Dog("Scout", 10);
Dog bailey = new Dog("Bailey", 5);

scout.feedDog();
"Gave Scout a bowl of food."

bailey.feedDog();
"Gave Bailey a bowl of food."
```

Private vs Public

Private

An instance variable or method that can only be accessed within the class

- On the AP Exam all instance variables should be private
- Some methods can be private if they are only used internally

Public

An instance variable or method that can be accessed outside of a class like in the main method

- Most methods are public

Object-Oriented Design

A design philosophy used by programmers when developing larger programs

1. Decide what classes you'll need to solve a problem
2. Define the data (instance variables) and functionality (methods) for the classes
3. Utilize classes and objects to solve your problem

Data Encapsulation

Data (instance variables) and the code acting on it (methods) are wrapped together in a single implementation and the details are hidden.

Data is safe from harm by keeping it private

5.2 & 5.3
Constructors, Comments, and Conditions

The Anatomy of a Constructor

Things to keep in mind:

- The Constructor name (in red) must **always** match the name of the class (in blue)
- Always prepend the **public** keyword before your constructor name
- Constructors have **no return type**. **Not even void!**
- The constructor definition is often included before other method definitions (but is not required)

```
public class ClassName
{

   // Instance Variable Declarations

   // Constructor - same name as Class, no return type
   public ClassName()
   {
     // Implementation not shown
   }

   // Other methods ...
}
```

Instance Variable Initialization

- Instance Variables are properties of your class (*name, age, and isAlive* in this example)
- They are normally given initial values **within** your Constructor
- They can also be defined in place (i.e. `private boolean isAlive = true;`)

```
public class Person
{
  private String name;
  private int age;
  private boolean isAlive = true;

  public Person(String initName, int initAge)
  {
     name = initName;
     age = initAge;
  }
 }
```

Types of Constructors

- **Default Constructor**
- No-Argument Constructor
- Parameterized Constructor
- Overloaded Constructors

```
public class Person
{
  // No constructor defined
  // Java creates Default Constructor
}
```

- Automatically generated by the Java compiler when you do not supply a Constructor
- Does not do any kind of specialized initialization of the Instance beyond whatever inplace variable initialization that exists
- Instance variables without inplace initialization will default to something "reasonable"
  - `int -> 0`
  - `boolean -> false`
  - `String -> null`
  - `Object -> null`

Types of Constructors

- Default Constructor
- **No-Argument Constructor**
- Parameterized Constructor
- Overloaded Constructors

```
public class Person
{
  private String name;
  private int age;
  public Person() {
    name = "Billy";
    age = 25;
  }
}
```

- A Constructor you define that takes no arguments
- Can perform any kind of initialization that the the Instance requires
- Every Instance is initialized exactly the same
- No values can be passed in during **`new`** to customize the Instance

When you define a Constructor (any kind) Java will NOT create a Default Constructor!

Types of Constructors

- Default Constructor
- No-Argument Constructor
- **Parameterized Constructor**
- Overloaded Constructors

```
public class Person
{
  private String name;
  public Person(String initName) {
    name = initName;
  }
}
```

- A Constructor you define that takes arguments
- Can perform any kind of initialization that the the Instance requires
- Each Instance is initialized individually based on the values passed into **`new`**

Types of Constructors

- Default Constructor
- No-Argument Constructor
- Parameterized Constructor
- **Overloaded Constructors**

```
public class Person
{
  private String name;
  private int age;
  public Person() {
    name = "Name unknown";
  }
  public Person(String initName) {
    name = initName;
    age = 30;
  }
  public Person(String initName, int initAge) {
    name = initName;
    age = initAge;
  }
}
```

- A Class may have multiple Constructors
- Each Constructor must be named the same as the Class; have no return type; and have a distinct set of parameters (types)
- These are useful when you want to provide default values for some Instance variables - While allowing other Instance variables to be set via `new`

Documentation is a love letter that you write to your future self.

- Damian Conway -

Comments

- Comments are a way for you to annotate your code
- This is text in your program that is never run by Java and is added for the benefit of the person reading the code
- You can also "comment out" a block of code during development to assist in the development or debugging process
- There are 3 ways to write comments in Java

Types of Comments

- **Single-Line Comment**
- Multi-Line Comment
- Documentation Comment

- A single-line comment starts with a double forward-slash (`//`)
- Can start anywhere - i.e. does not need to be in column 0
- All characters following the double forward-slash are ignored until newline or end of file

```
thisCodeWillRun();
// thisCodeWillNotRun();
thisCodeWillRun();

thisCodeWillRun(); // woo-hoo!
```

Types of Comments

- Single-Line Comment
- **Multi-Line Comment**
- Documentation Comment

- A multi-line comment starts with a forward-slash asterisk (`/*`)
- Can begin anywhere - i.e. does not need to be in column 0
- All characters - including newlines - are considered part of the comment until a asterisk forward-slash (`*/`) is encountered
- Your editor may have a key command that automatically converts a block of code into a multi-line comment

```
/*
thisCodeWillNotRun();
thisCodeWillNotRun();
*/
```

```
/* thisCodeWillNotRun(); */
```

```
thisCodeWillRun(); /* woo-hoo! */
```

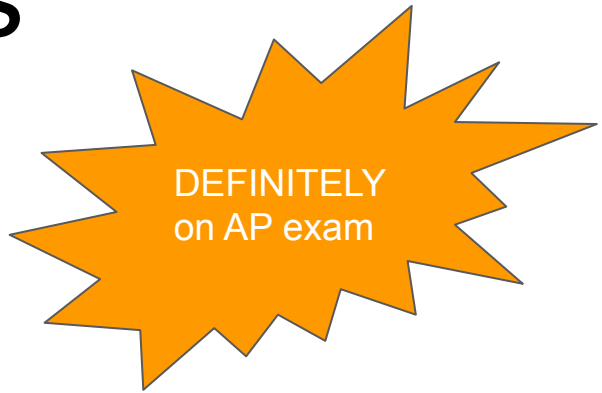
Types of Comments

- Single-Line Comment
- Multi-Line Comment
- **Documentation Comment**

- A variant of the multi-line comment syntax - Documentation Comments start with a forward-slash asterisk asterisk (`/**`)
- Typically found just prior to the definition of a function or method
- All characters - including newlines - are considered part of the Documentation Comment until a asterisk forward-slash (`*/`) is encountered
- Within a Documentation Comment - other standard components may be supported

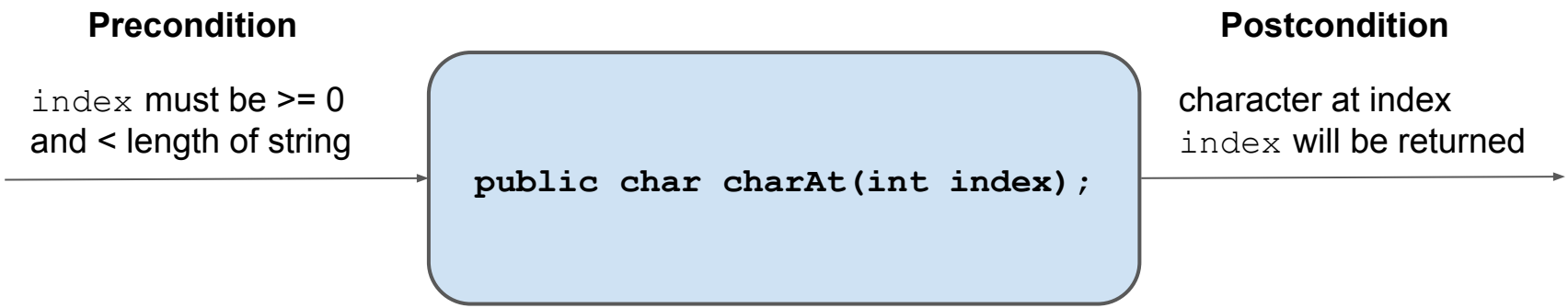
```
/**
 * Documentation comment
 *
 */
myMethod()
```

Preconditions and Postconditions



Preconditions and Postconditions

Preconditions and **postconditions** are a "contract" that describes what a method requires about its inputs, and what it promises as output.



Preconditions

Preconditions are part of the method's documentation, and may exist only as comments.

There is no expectation that the method will check to ensure preconditions are satisfied.

They may or may not be enforced by the method's code – the programmer using the method should read the documentation and understand the "contract" the method offers.

```
/**
 * Precondition: num2 is not zero.
 * Postcondition: Returns the quotient of num1 and num2.
 */
public double divide(double num1, double num2)
{
   return num1 / num2;
}
```

Who enforces preconditions?

Sometimes preconditions ARE enforced by the method's code.

An actual implementation of Java's `String.charAt`:

```
public char charAt(int index) {
      if ((index < 0) || (index >= value.length)) {
            throw new StringIndexOutOfBoundsException(index);
      }
      return value[index];
}
```

Here, an exception is thrown if the precondition is not satisfied.

Throwing an exception in Java is a common way to handle failed preconditions.

Who enforces preconditions?

Sometimes preconditions are enforced by some other mechanism.

`str.indexOf(null)` doesn't work, so `String.indexOf(String str)` has a precondition that `str` is not `null`.

```
jshell> "hello".indexOf(null)
|  Exception java.lang.NullPointerException: Cannot invoke "String.coder()" because "str" is null
|        at String.indexOf (String.java:2503)
|        at (#8:1)

jshell>
```

The programmer of `indexOf` decided it was OK to let Java throw a `NullPointerException` to "enforce" the precondition.

Depending on the situation, it may or may not make sense to enforce preconditions in code... but they definitely should be documented.

Who enforces preconditions?

At other times, it is not reasonable for the method to enforce the precondition.

The programmer calling the method must understand the preconditions and satisfy them.

The precondition here exists only as documentation, and describes the consequences of failing to meet it.
(Where is the precondition?)

binarySearch

```
public static int binarySearch(int[] a,
                               int key)
```

Searches the specified array of ints for the specified value using the binary search algorithm. The array must be sorted (as by the `sort(int[])` method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

a - the array to be searched

key - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, `(-(`*insertion point*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Postconditions

A **postcondition** is a condition that is true after running the method. It is what the method promises to do.

Postconditions describe the outcome of running the method, for example what is being returned or the changes to the instance variables.

Examples:

- **`String.compareTo()`** The method returns 0 if the string is equal to the other string. A value less than 0 is returned if the string is less than the other string (less characters) and a value greater than 0 if the string is greater than the other string (more characters).
- **`Math.random`** Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

Javadoc

Documentation for Java code is often written as documentation comments, and then HTML is generated with a tool called `javadoc`.

Code and documentation used to be written separately... the idea of doc comments is that if the doc is with the code, it's less likely to be stale.

The javadoc often describe a method's preconditions and postconditions.

```
/**
 * Returns the {@code char} value at the
 * specified index. An index ranges from {@code 0} to
 * {@code length() - 1}. The first {@code char} value of the sequence
 * is at index {@code 0}, the next at index {@code 1},
 * and so on, as for array indexing.
 *
 * <p>If the {@code char} value specified by the index is a
 * <a href="Character.html#unicode">surrogate</a>, the surrogate
 * value is returned.
 *
 * @param      index   the index of the {@code char} value.
 * @return     the {@code char} value at the specified index of this string.
 *             The first {@code char} value is at index {@code 0}.
 * @throws     IndexOutOfBoundsException  if the {@code index}
 *             argument is negative or not less than the length of this
 *             string.
 */
public char charAt(int index) {
    if (isLatin1()) {
        return StringLatin1.charAt(value, index);
    } else {
        return StringUTF16.charAt(value, index);
    }
}
```

5.4 & 5.5
Accessor Methods and Mutator Methods

Creating Classes (recap)

- **Instance Variables & Methods**
- **Public vs Private**
- **Constructors**
- Accessor Methods
- Mutator Methods

- Instance Variables (a.k.a attributes, properties, fields) hold the data of an Object and Instance Methods define the behaviors of an Object
- `public` instance variables and methods are **accessible from outside** an Object
- `private` instance variables and methods are **only accessible from within** the Object
- Constructors define how a Class is initialized when it is created with `new`

Creating Classes

- Variables & Methods
- Public vs Private
- Constructors
- **Accessor Methods**
- Mutator Methods

- `public` methods used to provide read-only access to `private` instance variables within the Object
- Sometimes these are called "get methods" or "getters"
- These `public` methods have a return type that matches the type of the `private` variable being returned

Creating Classes

- Variables & Methods
- Public vs Private
- Constructors
- **Accessor Methods**
- Mutator Methods

- Accessor methods are commonly named `get+VariableName` and do not have parameters

```
public class Person
{
  private String name;

  public Person(String initName) {
    name = initName;
  }

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

Creating Classes

- Variables & Methods
- Public vs Private
- Constructors
- **Accessor Methods**
- Mutator Methods

- Accessor methods can also return a filtered or transformed `private` variable value

```
public class Person
{
  private String phoneNumber;

  public String getAreaCode() {
    return phoneNumber.substring(0,3);
  }
}
```

Creating Classes

- Variables & Methods
- Public vs Private
- Constructors
- Accessor Methods
- **Mutator Methods**

- `public` methods used to modify internal `private` instance variables
- Sometimes these are called "set methods" or "setters"
- These `public` methods typically have a `void` return type and a parameter that matches the type of the `private` instance variable being modified

Creating Classes

- Variables & Methods
- Public vs Private
- Constructors
- Accessor Methods
- **Mutator Methods**

- Mutator methods are commonly named `set+VariableName` and have a single parameter

```
public class Person
{
  private String name;

  public Person(String initName) {
    name = initName;
  }

  public String getName() {
    return name;
  }

  public void setName(String newName) {
    name = newName;
  }
}
```

Creating Classes

- Variables & Methods
- Public vs Private
- Constructors
- Accessor Methods
- **Mutator Methods**

- Mutator methods can also filter, verify, or transform a value before assigning it to a private instance variable value

```
public class Person
{
  private String areaCode;

  public void setAreaCode(String phoneNumber) {
    areaCode = phoneNumber.substring(0,3);
  }
}
```

5.6: Writing Methods

Methods

- We have already covered about HOW to create Methods - but we have not spent much time talking about WHEN you should consider moving code into a Method
- Some of the WHENs
  - You have the same (or very nearly the same) block of code written in multiple places
  - You want to reduce complexity (improve development velocity / reduce code brittleness)
  - You want to write tests for a block of code
  - You have Methods that are excessively long (more than a single page)

Methods: Repeated Code

```
public class Person {

  private String firstName;
  private String lastName;

  public Person(String fn, String ln) {
    firstName = fn;
    lastName = ln;
  }

  public void sayHello() {
    String fullName = firstName + " " + lastName;
    System.out.println("Hello " + fullName);
  }

  public void sayGoodbye() {
    String fullName = firstName + " " + lastName;
    System.out.println("Goodbye " + fullName);
  }
}
```

```
public class Person {

  private String firstName;
  private String lastName;

  public Person(String fn, String ln) {
    firstName = fn;
    lastName = ln;
  }

  public void sayHello() {
    System.out.println("Hello " +  fullName());
  }

  public void sayGoodbye() {
    System.out.println("Goodbye " +  fullName());
  }

  private String fullName() {
    return firstName + " " + lastName;
  }
}
```

Methods: Reduce Complexity

```
public class FenceMaintenance {
 private int fenceWidth, fenceHeight;
 public FenceMaintenance(int width, int height) {
   fenceWidth = width; fenceHeight = height;
 }
 public paintFence() {
   int paintBucketsNeeded = (fenceWidth * fenceHeight) /
10;
   int brushesNeeded = paintBucketsNeeded * 1.5;
   double totalCost = 1.50 * brushesNeeded;
   totalCost += 12.99 * paintBucketsNeeded;
   System.out.println("Supplies purchased for $" +
totalCost);
   int totalTime = fenceArea / 5;
   System.out.println("Time required is " + totalTime +
"minutes.");
 }
}
```

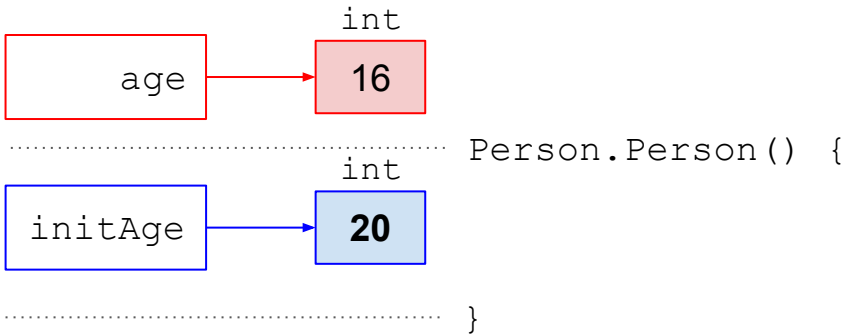
```
public class FenceMaintenance {
  private int fenceWidth, fenceHeight;
  public FenceMaintenance(int width, int height) {
    fenceWidth = width; fenceHeight = height;
  }
  public paintFence() {
    System.out.println("Supplies purchased for $" + calcCosts());
    System.out.println("Time required is " + calcTimeNeeded() +
"minutes.");
  }
  private int calcPaintBucketsNeeded() {
    return calcFenceArea() / 10;
  }
  private int calcFenceArea() {
    return fenceWidth * fenceHeight;
  private double calcCosts() {
    return calcBrushesNeeded()      * 1.50 +
           calcPaintBucketsNeeded() * 12.99;
  }
}
```

```
private int calcBrushesNeeded() {
  return calcPaintBucketsNeeded() *
1.5;
}
private int calcTimeNeeded() {
  return calcFenceArea() / 5;
}
```

Method Parameters: **Pass by Value** / Pass by Reference

- Parameters passed into Methods behave differently if they are primitive types or Object references
- Primitive Types (byte, short, int, long, float, double, boolean, char)
  - A copy is made when the Method is invoked for use during the life of the Method
  - The Method cannot alter the value of the variable (passed as a parameter) that exists in the caller of the Method

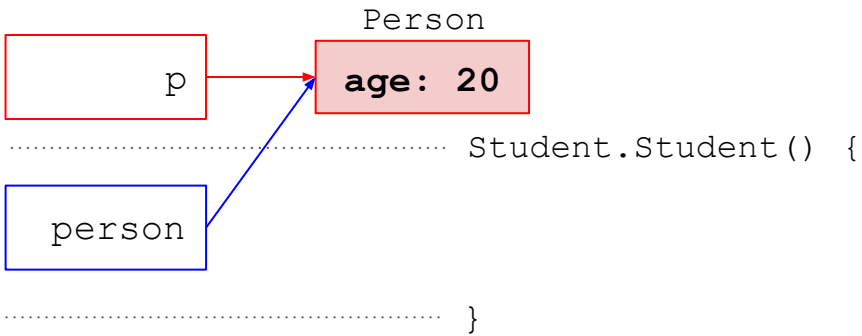
```
int age = 16;
Person p = new Person(age);
// age is guaranteed to still be 16


public class Person {
  public Person(int initAge) {
    // initAge is a COPY of age
    initAge = 20; // does not alter the value of age in the caller
  }
}
```



Method Parameters: Pass by Value / **Pass by Reference**

- Object Types / Classes
  - A reference to the Object is passed into the Method
  - The Method can alter the internal value of the Object passed as a parameter; And the caller can see these changes when it access the Object

```
int age = 16;
Person p = new Person(age);
// p.age == 16
Student s = new Student(p);
// p.age == 20
```




```
public class Student {
  public Student(Person person) {
    // person is a REFERENCE to the same object in the caller
    person.age = 20; // alters the Person passed in from the the caller
  }
}
```

Method Parameters: Pass by Value / **Pass by Reference**

- Object Types / Classes
  - A reference to the Object is passed into the Method
  - The Method can alter the value of the variable (passed as a parameter) that exists in the caller of the Method - **RARELY USED / NOT A BEST PRACTICE**

```
Person p = new Person();
p.age = 16;
Student s = new Student(p);
// p.age now equals 20 - LIKELY UNEXPECTED BEHAVIOR

public class Student {
  public Student(Person person) {
    // person is a REFERENCE to the same object in the caller
    person.age = 20; // alters the Person passed in from the the caller
  }
}
```

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

- **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 directlyaccess 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> Julie is one of 2 people

p1.getNumPeople();

C> 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 String email;

  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 String email;

  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 String email;

  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;
  }
}
```

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");
  }
}
```