11/7/22

5.2 & 5.3
Constructors, Comments, and Conditions

Recap

- Back in Unit 2, we learned how to create objects using a constructor:

```
// To create a new object, write:
// ClassName variableName = new ClassName(arguments);
World world = new World();
Turtle t = new Turtle(world);
Person p = new Person("Pat","pat@gmail.com","123-456-7890");
```

- Now, we're going to learn how to *define* Constructors

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

- 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`

Default Constructor

```
public class Person
{
  public String name;
  public int age = 15;

  // No constructor defined
  // Java creates Default Constructor
}
```

```
Person person = new Person();

System.out.println(person.name);
> null

System.out.println(person.age);
> 15
```

Types of Constructors

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

- 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

No-Argument Constructor

```
public class Person
{
  public String name;
  public int age = 15;

  public Person() {
    name = "Billy";
    age = 25;
  }
}
```

```
Person person = new Person();

System.out.println(person.name);
> Billy

System.out.println(person.age);
> 25
```

No-Argument Constructor

```
public class Person
{
  public String name;
  public int age = 15;

  public Person() {
    name = "Billy";
    age = 25;
  }
}
```

```
Person person = new Person();

System.out.println(person.name);
> Billy

System.out.println(person.age);
> 25
```

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

- 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`

Parameterized Constructor

```
public class Person
{
  public String name;
  public int age = 15;

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

```
Person person = new Person("Julie");

System.out.println(person.name);
> Julie

System.out.println(person.age);
> 15
```

Parameterized Constructor

```
public class Person
{
  public String name;
  public int age = 15;

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

```
Person person = new Person("Julie");

System.out.println(person.name);
> Julie

System.out.println(person.age);
> 15
```

Question: What happens here? And why?

```
Person otherPerson = new Person();
```

Parameterized Constructor

```
public class Person
{
  public String name;
  public int age = 15;

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

```
Person person = new Person("Julie");

System.out.println(person.name);
> Julie

System.out.println(person.age);
> 15
```

Question: What happens here? And why?

```
Person otherPerson = new Person();
** ERROR **
```

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**

- 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`

Overloaded Constructors

```
public class Person
{
  public String name;
  public int age = 15;

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

```
Person person1 = new Person();
System.out.println(person1.name);
A>
System.out.println(person1.age);
B>

Person person2 = new Person("Julie");
System.out.println(person2.name);
C>
System.out.println(person2.age);
D>

Person person3 = new Person("Julie", 25);
System.out.println(person3.name);
E>
System.out.println(person3.age);
F>
```

Overloaded Constructors

```
public class Person
{
  public String name;
  public int age = 15;

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

```
Person person1 = new Person();
System.out.println(person1.name);
A> Name unknown
System.out.println(person1.age);
B> 15

Person person2 = new Person("Julie");
System.out.println(person2.name);
C>
System.out.println(person2.age);
D>

Person person3 = new Person("Julie", 25);
System.out.println(person3.name);
E>
System.out.println(person3.age);
F>
```

Overloaded Constructors

```
public class Person
{
  public String name;
  public int age = 15;

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

```
Person person1 = new Person();
System.out.println(person1.name);
A> Name unknown
System.out.println(person1.age);
B> 15

Person person2 = new Person("Julie");
System.out.println(person2.name);
C> Julie
System.out.println(person2.age);
D> 30

Person person3 = new Person("Julie", 25);
System.out.println(person3.name);
E>
System.out.println(person3.age);
F>
```

Overloaded Constructors

```
public class Person
{
  public String name;
  public int age = 15;

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

```
Person person1 = new Person();
System.out.println(person1.name);
A> Name unknown
System.out.println(person1.age);
B> 15

Person person2 = new Person("Julie");
System.out.println(person2.name);
C> Julie
System.out.println(person2.age);
D> 30

Person person3 = new Person("Julie", 25);
System.out.println(person3.name);
E> Julie
System.out.println(person3.age);
F> 25
```

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! */
```

Start Multi-Line Comment
Roll your fingers from the left to the right

End Multi-Line Comment
Roll your fingers from the right to the left

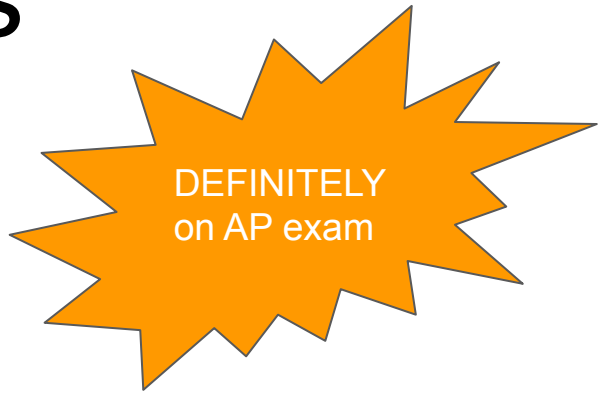
Types of Comments

- 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

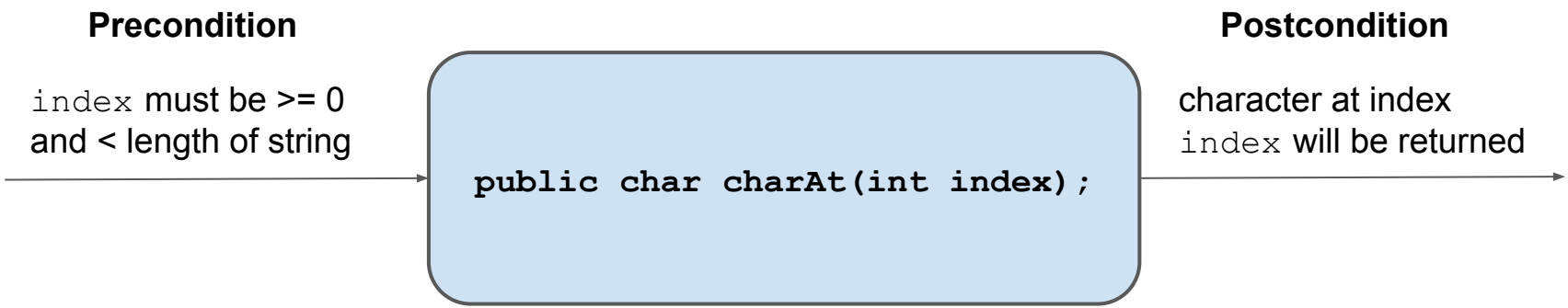
```
/**
 * Documentation comment
 *
 */
myMethod()
```

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);
    }
}
```

Practice on your own

- CSAwesome 5.2 Writing Constructors
- CSAwesome 5.3 Comments and Conditions
- ConditionWorld Exercise in Replit