Java classes, part 2

Prof Bill, Jan 2020

It's...Java classes, cont. These notes are a potpourri of smaller Java features. I'm just tying some loose ends about Java classes.

thanks...yow, bill

Our "textbook" links:

- Oracle Java, docs.oracle.com/javase/tutorial/java
- Sedgewick Java, introcs.cs.princeton.edu/java/home
- Sedgewick, Algorithms, algs4.cs.princeton.edu/home

Thank you - I bow down to Noctrl's own Dr. Godfrey Muganda for his excellent Java textbook. I have liberally borrowed ideas from this book.

media.pearsoncmg.com/bc/abp/cs-resources/products/product.html#product,isbn=0134038177

Static class members

Usually, classes have **instance variables/methods**, which are accessed/called for a specific object of that class. That's almost always the setup.

A class can also have **static variables** or **static methods**. These are accessed/called for the class, rather than one specific object of the class.

Example 1: your main() method is always static.

Example 2: The Math class in the Java API is a good example. All the methods in the class are static. docs.oracle.com/javase/8/docs/api/java/lang/Math.html

```
// Math class example: static double max(double a, double b)
double test1, test2, value;
// some code...
value = Math.max( test1, test2);
```

Notice: Use the class to prefix a static member. You can also use any object of the class, but the class name is preferred, imho.

~~> google: java static variable; java static method; java static member

---

References to objects

Objects are always stored as references (or "pointers"). Examples: variables, method arguments, and return values from methods.

Objects are held by reference, not copied. A couple examples:

```
Bubble b1 = new Bubble( 3.14);
Bubble b2 = b1; // not a copy
Bubble b3; // without assignment, var is null

b1.popBubble( 3); // popBubble method uses reference to b
```

~~> google: java object reference

The toString() method

The toString() method is defined in the Java Object class. So, it is available to every object in your program.

```
public String toString();
```

docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Java automatically calls toString() to get a String for any Object.

```
Dog myPet;
// some code...
System.out.println( myPet); // calls toString() in Dog class
```

By default, toString() returns an object's internal pointer, which isn't very helpful.

Often, you will **override** toString() for your class so that it's more helpful:

```
public class Customer {
  // some code...

  public String toString() {
    // lastName and firstName are instance variables in Customer
    return lastName + ", " + firstName;
  }
}

/* now, my toString method is called automatically by Java */
Customer c = new Customer( "Jimmy", 200);
System.out.println( "Fave customer is " + c);
```

String tip of the day (STOTD):

- Strings are immutable and can never be changed; so each String assignment or operation creates a new String.
- This can be inefficient in building larger, more complex strings.
- The **StringBuilder** class is a good way to efficiently build complex strings. You'll see it used in toString() methods quite a bit. Java API: docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

~~> google: java toString

Writing an equals() method

The **== operator** compares pointers, not values in an object.

```
String s1 = new String( "Bill");
String s2 = s1;
if( s1 == s2) { /* yes, this is true */ }

String s3 = new String( "Bill"); // same value string
if( s1 == s3) { /* no, this is false! */ }

// use equals() method to compare values
if( s1.equals( s3)) { /* true! */ }
```

Like toString(), every object has an **equals()** method.

```
boolean equals(Object obj);
```

docs.oracle.com/javase/7/docs/api/java/lang/Object.html

By default, equals() compares pointers, ala the == operator, but you can **override** it.

```
public class Student {
  // code here...

  // two students are equal if their ID's are the same
  public boolean equals( Student s) {
    return this.studentId == s.studentId
  }
}

// example usage
Student s1, s2;
/* code here… */
if( s1.equals( s2)) { /* code here... */ }
```

~~> google: java equals method

Copy constructors

We know...**ctor** (constructor) is a method with the same name as the class. It is called when the new keyword is used to create an object.

There's a special case of this called a **copy constructor**. This method has the same name as the class and accepts an object as a parameter. The return value is a copy of the object passed in.

```
public class Student {
  // code here...

  public Student( Student copyMe) {
    this.name = copyMe.name;
    this.studentId = copyMy.id;
  }
}

// example call
Student sophomore;
/* code here… */
Student newbie = new Student( sophomore);
```

~~> google: java copy constructor

The this reference variable

In Java, **this** is a keyword. It references the current object in a class whose method is being called. "this" is used to avoid naming conflicts and make code clearer.

```
public class Monkey implements Animal {
  // fields
  private String name;
  private int iq;

  public Monkey( String name, int iq) {
    // set fields of this, the current object
    this.name = name;
    this.iq = iq;
  }
}
```

~~> google: java this keyword

Java garbage collection

In Java, we create new objects with the **new** keyword. We don't, however, explicitly destroy objects. This happens auto-magically.

Java has a **garbage collector**. It's part of the Java Virtual Machine (JVM) that runs your program. The garbage collector automatically reclaims the memory of objects that are no longer referenced.

Some languages (C, C++) require you to keep track of your objects and explicitly destroy them.

There's a tradeoff with garbage collection:

- Plus - it's awfully nice to not worry about destroying objects in your code
- Minus - garbage collection can be expensive

~~> google: java garbage collection

Important: *From here on…these are my notes from two sources:*

- ➢ *Sedgewick Algorithms 1.2 Data Abstraction*
- ➢ *Oracle Java - Object-Oriented Programming Concepts*

Some of this is a repeat of my previous notes. "It's a good thing." - Martha Stewart

---

ADT/OOP

Abstract data type (ADT) - a data type whose internal representation is hidden from the client.

Applications programming interface (API) - is an interface between different parts of a computer program intended to simplify its implementation and maintenance; often defined as a set of methods and variables

Object-oriented programming (OOP) - a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

Yes, these concepts combine to make software easier to create and maintain:

- Provide functionality
- While hiding the complexity of its implementation

The goal of OOP = manage complexity to increase productivity of software designers

Terms:

class - a blueprint from which individual objects are created

object - instance of a class

inheritance, single inheritance, multiple inheritance

interface - code describing an API (methods); like a contract

data hiding

method signature - the name parameters and return value of a method; this is the interface, not the body/code

design pattern - a common or "typical" solution to a design problem

polymorphism means “many forms” (example: Pet p = new Dog(“Brownie”);)

inheritance = is-a relationship; **aggregation** / **composition** = has-a relationship

For OOP, use public methods and private variables. Why?
Java only supports **single inheritance**. But not multiple inheritance. Why?

Convention: In class, instance variables are private; **accessor**/getter and **mutator**/setter methods are public.

Java intro, cont

Terms

keywords - reserved words in a programming language

Java Virtual Machine, compiler, executable code

IDE - Integrated Development Environment; a fancy editor

variable, literal, primitive data types (int, float, char…)

constructor (ctor), accessor (getter), mutator (setter)

new operator - returns a reference to a newly-created object

instance variable, **class variable** (static)

method signature - name + parameter list of a method, it’s interface!

Simple console I/O:

- **System.out** is a **PrintStream** object, includes print() and println() methods
- Read from input stream using **Scanner** class with **System.in**

Strong **console** I/O idioms:

```
Scanner keyboard = new Scanner(System.in);   // input
String username = keyboard.nextLine();
```

```
System.out.println( “This is fun.”);   // output
```

Comment styles:

```
/* comments ignored by the compiler */

// end of line comment

/**
 * Javadoc comment!
 **/
```

javadoc - commenting convention used by all JDK code, we *must* use!

Strings:

- ❏ part of standard Java API/library
- ❏ **immutable**
- ❏ comparison methods: equals(), equalsIgnoreCase(), compareTo()
- ❏ Use **StringBuilder** to manipulate strings.

Modifiers:

- **public**, **private**, **protected** - controls visibility to class variables and methods
- **abstract** - defines an interface, but no body/code
- **static** - makes a class variable or method (rather than instance)
- **final** - for variable, an initial value can never be changed, a constant; for method, this means it cannot be overridden

Small stuff:

- logical operators: and (&&), or (||), not (!)
- loops: while, for, do-while; break, continue stmts within a loop
- Increment (x++) and decrement (y--)
- Java uses **Unicode** for char representation (2 bytes)
- **Random** class, Java.Util.Random - generate random numbers

abstract class - in between concrete class and interface, some methods are abstract; we will rarely use these, but JCF does

exceptions - try, catch, throw, throws; exception hierarchy

generics - replace Object because “code became rampant with such explicit casts”

ctor rules are complex: default ctor, ctor overloading, super, this, etc

Etc

Agile: The Programming Process… agile, agilemanifesto.org

Agile is (sort of) the opposite of top-down planning. The agile rules are tiny.

> Individuals and interactions over processes and tools
>
> Working software over comprehensive documentation
>
> Customer collaboration over contract negotiation
>
> Responding to change over following a plan

We (CSC 210) are agile.

UML class diagram - a quick way to communicate class variables and methods

What is the **UML** representation for class, attributes, is-a relation, has-a relation? (see p 65) The relations between classes is a critical design decision.
Some nice text/examples in Wikipedia: en.wikipedia.org/wiki/Class_diagram

Java has a strong idiomatic **programming style**: camel notation, indentation, always use curly braces with if and loops, etc. Use Javadoc for comment blocks.

Here's a nice intro to Javadoc, www.tutorialspoint.com/java/java_documentation.htm.

Coding conventions:

- Code Conventions for the Java TM Programming Language (by Sun Micro, really old), www.oracle.com/technetwork/java/codeconvtoc-136057.html
- Google Java Style Guide, google.github.io/styleguide/javaguide.html
- **Prof Bill's** Java Coding Guidelines, wtkrieger.faculty.noctrl.edu/etc/java_coding_guidelines.pdf