CSC 216 Intro: Basics of What You Should Already Know

Java Language, OOP, Classes and Objects, Access, Constructors, Variables and Instances, null, Methods, Parameters, this, Lifetime, Scope, Packages

CSC 216 topics you should know

- Constructing/compiling/running Java programs
- Class fundamentals and OOP basics
- Primitive and String types
- Methods and parameters
- Control structures: conditions, loops
- Arrays and array processing
- GUIs (some)
- Scanners and Text file processing
- Exceptions – try/catch for built in exception types

Java and the Java VM

- Running Java requires the JRE, which includes:
  - JVM
  - Java API and code libraries
  - Java applet viewer
- JDK is required to develop Java programs
- Which Java? 1.8 or later.
- Java API is large library of built-in code for language basics (strings, wrapper classes, etc), applets, GUIs, I/O, security, utilities, ….
- API is fully documented.
- Java applications can be:
  - Desktop: Console or GUI
  - Applets (run in a browser)
  - Web based: Servlets (server side apps)

Object-oriented programming

- **OOP** - Programs are models of problems that they solve. Models consist of interacting objects.
  Objects in a model have:
  - **Attributes** (or state). Internal data of an object (fields or **instance variables**).
  - **Behaviors**. Set of actions that the object can perform (instance methods).
  - **Identities** (names/locations).
- **Class** - description of a type of object.
- **Objects** - instances of classes.
- **Variable** - a reference to an object.
- Objects interact by sending each other **messages** (method calls).

Class definitions

```
package edu.ncsu.csc216.samples;
public class Account {
   private String owner;
   private double balance;
   public static final double CHECK_FEE = .15;

   public Account(String owner, double balance) {
      setOwner(owner);
      if (balance < 100)
         balance = 100;
      this.balance = balance;
   }

   public Account (String owner) {
      this(owner, 100);
   }

//... Continued next page
```

Class definitions (continued)

```
  public String getOwner() {
      return owner;
   }

   public double getBalance() {
      return balance;
   }

   public void setOwner(String name) {
      if (name == null || name.equals(""))
         throw new IllegalArgumentException("Empty name");
      owner = name;
   }

   // methods to write a check or make a deposit
}
```

Access

- Access to class members can be:
  - **private**: accessed only inside the class definition
  - **public**: accessed by any code
  - **protected**: accessed by code in the same package or by descendant classes
  - **default**: accessed only in the same package (also known as package-private)

Modifier	Class	Package	Subclass	World
None	Y	Y	N	N
public	Y	Y	Y	Y
Private	Y	N	N	N
Protected	Y	Y	Y	N
None	Y	Y	N	N

Constructing objects

- **Constructor**
  - Purpose is to create the object in an acceptable state.
  - A constructor call is required to create an object. Calls use the keyword **new**.

```
    Account a = new Account(“Jane Doe”, 235.4);
    Thermometer t = new Thermometer();
```

- **Null constructors** have no parameters.
- Classes can have multiple constructors
  - Different signatures – number and type(s) of parameters
  - Which constructor is called depends on the actual parameters
- Every class has at least one constructor. If you don’t define one, Java automatically creates a **default constructor**, which is null. If you do define a constructor, Java does not provide the default.
- Programming Paradigm: one constructor contains true initialization code – all other constructors call it. This ensures that every object goes through a common code path.

Object references/variables

A (class type) **variable** is a **reference** to an object. (It holds the address of the object.)

```
Account a = new Account(“Jane Doe”, 235.4);
```

- Creates an `Account` reference named `a`.
- Creates an `Account` object.
- Calls one of the `Account` constructors.
- Assigns the address of the newly created object to `a`.

```
Account b = a;
```

- Creates a new reference to an existing object.

Dereferencing

```
Account x = new Account(“Ralph Doe”, 235.4);
...
String s = x.getOwner().toUpperCase();
x.setOwner(“John Doe”);
```

Dereferencing an object means accessing data or methods of the object.

- Uses the dot operator.
- Requires the object (through a reference such as `x`) and the method or field (`getOwner()` or `setOwner()`).

Null References

null is a built in value that does not refer to any object.

A null reference is an attempt to dereference a variable that is null. The following will generate a **NullPointerException**:

```
String y;
String z = y.toUpperCase();
```

You can work with null:

- Initialize a variable.

```
Account d = null;
```

- Check before trying to dereference:

```
if (a == null) {  // etc
…
if (a != null) {  // etc etc
```

- Return from a method. Often indicates failure.

Methods in the context of OOP

- Object behaviors occur in response to messages.
- A **message** is a call to a class method via an object. Format is <reference>.<message>(<actual parameters>)
- Simple messages:
  - **Getter** – returns the value of an attribute.
  - **Setter** – sets the value of an attribute.
  - **Mutator** – any method that can change an attribute value (not necessarily simple).
- **Parameters** – information sent with a message.

  Parameters, which are either references or primitives, are "passed by value." The method works with a copy of each reference parameter (rather than a copy of the object) and a copy of each primitive parameter.
- **Overloading** – two methods in the same class with the same name. The signatures (parameter lists) must be different.

Objects as parameters

```
Account act = new Account(“Jane Doe”, 235.4);
...
addInterest(act);
...
public void addInterest(Account p) {
```

- `act` is the **actual parameter** for the call.
  `p is the formal parameter in the method.`
- `p` becomes another reference the object pointed to by `act`.
  Parameters that are references become different names for the object inside the method code.
- `act` does not change – it contains the same reference as before the call.
- The fields of `act’s` object may change as a result of the call.

this

- Keyword used in non-static methods.
- Implicit parameter – a name inside a method for the object that is receiving the message.
- Critical for dealing with potential naming conflicts.
- Example:

```
public class C {
   private int x;
   // …
   public void setX(int x) {
       this.x = x;
   }
```

Lifetime

- **Lifetime** (of a variable) - the the extent of time during execution that memory is devoted to the variable. Applies to objects and primitives.
- In the context of a method
  - local variable lifetime is the duration of method execution
  - formal parameter lifetime is duration of method execution
- Lifetime of an instance variable is the lifetime of the object it belongs to.
- Object lifetime is from the point of creation until all references to the object are gone.

Scope of identifiers

- **Scope** (of an identifier) – the section of code where the identifier is recognized as declared.
- Scope kinds:
  - *Class* - class member = the entire class.
  - *Local* - local variable or parameter = the method where declared
- Cannot have two variables identifiers with the same scope and same name.
  - Two variables declared in the same method must have different names.
  - A local variable cannot have the same name as a formal parameter.
  - Two instance variables cannot have the same name.

Hiding

- **Hiding** - when an attribute (instance or class variable) has the same name as a local variable or parameter.
- Example:

```
public class MyClass {
   private int x;
   private int count;
   private static int s;
   // …
   public void method(int x, int s) {
      int count = 0;  // local variable count hides
                      //   instance variable count
       x = 17 ;   // parameter x hides instance variable x
       s = count + x; // parameter s hides static variable s
   }
```

- To reveal attributes:
  - Use `this` as in `this.count`
  - Use class name as in `MyClass.s`

Static variables and methods

- **Static** variables and methods – members that belong to the class rather than a particular instance of the class.
  - Also called “class variables” and “class methods.”
  - Can be accessed via the class without reference to an instance of the class.
  - Used for class-wide data or methods that do not require any instance variable information.
  - Declared with keyword static
- Example: Math.PI from the Math class.
- Example:

```
public class Account {
   private static double fee;
   private static final double PENALTY = 5.0;
   // …
   public  static void setFee(double fee) {
       Account.fee = fee;
   }
```

Packages

- **Package** – programmed grouping of related classes and interfaces.
  - provide *namespaces*, so different classes in different packages can have the same name.
  - can be archived into JAR files, useful for distributing Java apps or libraries.
- Package statement must be the first statement in a program.
- Default package = no package statement.
- Packages can be nested. Use the dot operator to distinguish inner package name from outer.
- Fully qualified class name is <package name>.<class name>.

```
javax.swing.border.EmptyBorder
```

Using packages

- Options:
  - Option 1: Use the fully qualified class name.
    - java.awt.Color myColor = java.awt.Color.RED;
    - java.util.ArrayList<String> names= new java.util.ArrayList<String>();
  - Option 2: Use import statements.
    - import java.awt.Color;
    - import java.util.ArrayList;
  - Option 3: Use import statements with wildcard:
    - import java.awt.*;
    - Import java.util.*;
- The compiler imports the java.lang package by default. Classes include:
  - Math
  - String
  - Object

Naming conventions

- Class Names. Nouns. Camel case, beginning with uppercase.
  - MyClass
- Instance and static variables. Camel case beginning with lowercase.
  - myVariable
- Constants (final, static). Uppercase with underscores
  - MY_CONSTANT
- Methods. Verbs. Camel case beginning with lowercase.
  - getValue()
- Boolean variables and methods: begin with *is.*
  - isValid(), isVisible
- Getters and setters: begin with *get* and *set*
  - getValue(), setValue()
- Package names. All lowercase, reverse url.
  - com.abc.application.ui
  - edu.ncsu.csc216.section601