Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 1: Introduction to Data Structures Course, Review of Object Oriented Programming and Advanced Java Topics from CSCI.UA 101.

Reading materials

Dale, Joyce, Weems: 1.2 – 1.6

Liang: 8,10,11, 15

Topics Covered

Joanna Klukowska
joannakl@cs.nyu.edu

Joanna Klukowska
joannakl@cs.nyu.edu

1 Introduction to Data Structures Course

"*I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*"

Linus Torvalds, 2006

1.1 What are data structures?

A **data structure** is a collection of data items organized in some fashion that allows items to be stored and retrieved by some fixed methods.

Examples of data structures from CSCI.UA 101:

- **array**: data items are organized so that they can be accessed by their index number,
- **stack**: collection of items with access (add/remove operations) only at the 'top' of the stack (think of a stack of plates).

Data structures covered in CSCI.UA 102:

- **(linked) list**
- **stack**
- **queue**
- **hash table**
- **tree**
- **binary tree**

Other data structures:

- many different types of trees
- many different types of graphs

1.2 What are algorithms?

An **algorithm** is a logical sequence of discrete steps that describes a complete solution to a given problem commutable in a finite amount of time and space.

Joanna Klukowska
joannakl@cs.nyu.edu

1.3 The Phases of Software Development (Software Engineering)

As programs grow larger and become group projects, rather than course assignments intended for individuals, the actual coding becomes a smaller part of the process of software life cycle. Problem analysis, algorithm development, software specification and design (choice of algorithms, data structures, division of tasks between programming teams, documentation, etc.), testing and maintenance become more and more important. The customers/clients and future users need to be involved at all stages of software development.

Software development is a process that involves several steps. Here is a list of typical software development phases:

1. Specification of the task.
2. **Design of a solution.**
3. Implementation (coding) of the solution.
4. **Testing and debugging.**
5. **Analysis of the solution.**
6. Maintenance and evolution of the system.

You have definitely been doing a lot of coding (step 3) in the earlier courses. In this course you will continue writing code, but you will also work on

- step 2: designing of the solution
  - How to solve the problem without a computer (you cannot code it, if you do not have a good understanding of what the solution is)?
  - How to represent various parts of the solution on a computer, i.e., in code?
  - What are the most common tasks and which data structures are the most appropriate?
  - What should the user interface look like? (This should involve future users inputs.)
- step 4: testing and debugging, once the code compiles and you run it, ...
  - Does it run according to the specification?
  - Does it run correctly on every input / every type of input?
- step 5: analyzing the code
  - Could the code be improved further? How? (in terms of user experience, performance: both memory and time)
  - What is the performance of the code?
  - Does it scale to much larger input sizes?

Joanna Klukowska
joannakl@cs.nyu.edu

Goals of Quality Software Quality software is much more than just code that compiles, runs, and (maybe) is correct. A good program should achieves the following goals:

- It **works** (and that means works correctly, according to the specification, provides usable user interface and is efficient).
- It **can be modified** without excessive time and effort by any software engineer/programmer, not just by its author.
- It is **reusable**.
- It is **completed on time** and within budget.

It's not easy to meet all these goals, but they are all important.

2 Method Specification/Documentation

One of the ways of ensuring code reusability and modifiability is to properly document your code. Another programmer should be able to ...

1. ... read your documentation and use your code (without ever reading your code itself). Think of all the classes and methods that come with Java API - you never read the actual source code for the `print` method, but you used it a lot.
2. ... develop the code for a method that performs the same task as your method does. Someone else's code might achieve the tasks in a different way, but it should accomplish the same final result. Your documentation should provide enough information to make it possible.

The specification should include the following information:

- the signature (method name, parameter list with types);
- short description;
- list of parameters with short descriptions;
- preconditions - list of conditions that have to be true for the method to function properly;
- postconditions / return value - meaning of the return value and any additional effects that the method has if all preconditions are true when it was called;
- list of exceptions that are thrown by the method.

The Javadoc style of comments makes it easy to generate this type of documentation if you follow certain patterns in your source code. The big advantage of using Javadoc is that you *kill two birds with one stone*: you document the source code and can easily generate user documentation.

Example

The following is the source code with Javadoc comments for a method that computes a volume of a sphere with specified radius:

```
1 /**
2  * Compute volume of a sphere with specified radius.
3  * precondition: radius >= 0
4  * @param radius
5  *     radius of a sphere, should be a positive floating point number
6  * @return
7  *     the volume of a sphere with specified radius
8  * @throws IllegalArgumentException
9  *     Indicates that radius parameter has illegal value.
10  */
11 public static double computeVolume ( double radius )
12 {
13     if ( radius < 0.0 )
14         throw new IllegalArgumentException ("Radius " + radius +
15                                             " should not be negative.");
16
17     return 4.0/3.0 * Math.PI * radius * radius * radius;
18 }
```

The documentation produced by running javadoc is as follows:

computeVolume

`public static double computeVolume(double radius)`
Compute volume of a sphere with specified radius.
precondition: radius >= 0
Parameters:
radius - radius of a sphere, should be a positive floating point number
Returns:
the volume of a sphere with specified radius
Throws:
java.lang.IllegalArgumentException - Indicates that radius parameter has illegal value.

2.1 Javadoc

In order to use Javadoc you need to do a few things:

- Use multiline comments by enclosing them in `/** ... */` symbols (rather than the usual `/* .. */` for multiline comments or // for single line comments).
- Use the regular Java comments: /* ... */ and // for all comments that SHOULD NOT be part of the documentation generated by Javadoc.
- Use tags (words preceded by an @ symbol) to indicate different types of information.
  - Some predefined tags are:
    - `@author [author name]`: identifies the author(s) of a class or interface.
    - `@param [parameter name] [parameter description]`: describes the parameters in a method or constructor.
    - `@return [description of what is returned]`: describes a return value from a method.

  - `@throws [list of thrown exceptions]`: specifies which exceptions might be thrown by the method and when

- Run `javadoc` on your source files to generate HTML documentation.
  - From command line you can simply run

    ```
    javadoc *.java
    ```

    in the directory that contains your source files.

    You can also use `-d dir_name` option to save the documentation to specified directory instead of the current working directory. (It is probably not a good idea to mix your Java source code files and HTML documentation files.) Running

    ```
    javadoc -d doc *.java
    ```

    will save the Javadoc HTML files to a `doc` subdirectory in a current working directory.
  - From Eclipse IDE you can go to *Project* menu and select *Generate Javadoc* option.
    - If you just want to create simple documentation that uses the default tags only, select *Finish* from the first dialog box that appears. (Notice that you can select which classes and methods should be included in the newly generated documentation.)
    - Eclipse by default creates a separate directory, called `doc`, for documentation of each project. You can change its location if you wish.

2.2 Code Example

Source code: Sphere.java

3 Object Oriented Design (Review)

An **object** represents an entity in the real world. For example: student, table, car, circle, university, book, bookstore.

Objects have:

- **state** represented by **data fields**
- **behavior** defined by **methods**

A **class** is a template for creating objects of the same type. For example, a `Circle` class can be used to create multiple `Circle` objects**.**

A **constructor** is a special kind of method that is used to construct an object. A class can have multiple constructors (i.e. different ways of creating objects).

Joanna Klukowska
joannakl@cs.nyu.edu

3.1 Reference Variables ⇔ Objects

The statement

```
ClassName objectRefVariable;
```

declares a reference variable that can be used to store a memory address at which the actual object is stored.

In order to create an object, you need to use the `new` operator

```
objectRefVariable = new ClassName(...);
```

This is similar to how an array-name stores the memory addresses of the location where the array is stored. The actual array storage needs to be allocated using the `new` operator.

As with all the other declarations/creation statements, the above two lines can be combined into a single statement

```
ClassName objectRefVariable = new ClassName(...);
```

NOTE: observe the parenthesis in the last two statements. This is a call to the constructor that may or may not take parameters.

Occasionally, you may want to create an object in memory (usually temporary object) that is not pointed to by any object reference variable. Those objects are called **anonymous objects**, since they do not have a name. For example

```
System.out.printf("The area of circle with radius %f is %f.  \n",
                                          5, new Circle(5).getArea() );
```

After this statement executes the object is still in the memory, but there is no way to access it. The memory that it occupies eventually gets reclaimed by Java garbage collection.

Common error Confusing object assignment with reference variable assignment.

```
Circle c1 = Circle(5);
Circle c2 = Circle(17);
c1 = c2;
System.out.printf("The radius of circle c1 is:  %f", c1.getRadius() );
System.out.printf("The radius of circle c2 is:  %f", c2.getRadius() );
c1.setRadius(25);
System.out.printf("The radius of circle c1 is:  %f", c1.getRadius() );
System.out.printf("The radius of circle c2 is:  %f", c2.getRadius() );
```

The third statement results in `c1` and `c2` pointing to the same Circle object in memory (the one with radius 17). It does not copy one Circle object to another. What will be printed when the above statements are executed?

Joanna Klukowska
joannakl@cs.nyu.edu

3.2 Object Composition (HAS-A relationship)

An object can contain another object as its data field. This relationship is called **has-a relationship**.

Example:

`Student` object **has-a**:

- name, which is a `String` object
- date of birth, which is a `Date` object
- address, which is an `Address` object
- ...

The "owner" object is called **aggregating object** (its class is **aggregating class**). The "subject" object is called **aggregated object** (its class is **aggregated class**).

3.3 Class Design Guidance

Cohesion

- The class should describe a single thing.

Consistency with Java programming style and conventions

- Place data fields before constructors and constructors before the other methods.
- Provide default (no-arg) constructors (if applicable).
- Use standard methods' and fields' names, for example `length`, `compareTo`, `toString`.
- Implement `toString` method.
- Implement `compareTo` method.

Encapsulation = Hiding implementation details

- Make all data fields private.
- Provide accessors (get methods) for fields that should be readable.
- Provide mutators (set methods) for fields that should be writable.
- Make helper methods private.
- Helper methods are the methods that should not be called from outside of the class.

Joanna Klukowska
joannakl@cs.nyu.edu

Clarity

- Provide easy to explain contract: methods should implement simple tasks.
- Methods should be independent, i.e., calling one method should not fail because another method has not been called first. Provide graceful way of quitting the method rather than letting it crash the program.
- Use intuitive meaning of names.
- Use independent data fields: Do not keep multiple data fields that can be derived one from another. Having data fields that are not independent implies that they all need to be modified when one of them changes. Exception to the rule: when computing the value is very costly.

Completeness

- Provide full and general functionality for the class (not only things you need in your next assignment). Think of all the different ways in which the class can be used.

3.4 Code Example

Source code: `Race01.java`, `RaceContender01.java`

This example provides a simple simulation of a race between two contenders that are instances of `RaceContender01` class. We will build on this code in later examples.

4 Inheritance and Polymorphism

4.1 Inheritance (IS-A relationship)

Inheritance

- defining new classes from existing ones;
- defining general classes that can be extended into more specialized classes.

Inheritance models **is-a relationship**. For example

- a student is-a person, so a `Student` class can inherit from a `Person` class;
- an employee is-a person, so an `Employee` class can inherit from a `Person` class;
- a professor is-an employee, so a `Professor` class can inherit from an `Employee` class (which itself inherits from a `Person` class);
- a student is NOT a professor (even though they might share many characteristics) so the two classes should not be related by inheritance relationship.

Joanna Klukowska
joannakl@cs.nyu.edu

If `C2` class extends `C1` class, `C2` is called a **subclass** and `C1` is called the **superclass**.

Defining superclasses and subclasses:

```
public class C1 {
    //class definition
}
public class C2 extends C1 {
    //class definition
}
```

The keyword `extends` indicates that the class `C2` **extends/inherits from** the class `C1`.

Class `C2` has access to all public and protected data fields and methods of class `C1`. It does not have access to private data fields and methods of `C1`.

NOTE: In Java a class can inherit from only one class, i.e., in the above example, `C2` cannot extend any other class than `C1`. This is called **single inheritance**. There are other programming languages in which multiple inheritance is allowed.

NOTE: Contrary to a misleading name, a subclass is not a subset of the superclass. In fact, it usually contains more than the superclass.

4.1.1 `super` keyword

The keyword `super` is used to access superclass constructors, methods and data fields (the ones that are inherited, not the private ones).

Examples:

`super();` invokes the default constructor of the superclass (assuming it exists). This call can be made only from within a constructor of a subclass.

`super(param);` invokes the one-parameter constructor of the superclass (assuming it exists). This call can be made only from within a constructor of a subclass.

`super.methodName(paramList);` invokes a method of the superclass. This call can be made from anywhere in the subclass.

`super.dataField;` accesses a dataField of the superclass. This call can be made from anywhere in the subclass.

4.1.2 Superclass methods and data fields

A subclass inherits all public and protected methods and data fields of its superclass. That means that a subclass can use the methods and data fields that it inherits without defining them.

Example:

Joanna Klukowska
joannakl@cs.nyu.edu

```
public class C1 {
    protected int x;
    public void setX ( int x ) {
        this.x = x;
    }
    //the rest of class definition
}
public class C2 extends C1 {
    protected float c;
    public C2 ( float c, int x ) {
        this.c = c;
        setX(x);
    }
    //the rest of class definition
}
```

In the above code, the class `C2` can access the data field `x` and the method `setX()` without defining them because they are inherited from `C1`.

The keyword `protected` is used to indicated data fields and methods that are accessible only from within the class and/or classes that inherit from it. Those data fields and methods are not accessible from outside of the class (they are `private` outside of subclasses).

One can use the keyword super to indicate the inherited methods but it is not necessary, unless the methods are overridden (see below). The class C2 above could be rewritten as

```
public class C2 extends C1 {
    protected float c;
    public C2 ( float c, int x ) {
        this.c = c;
        super.setX(x);
    }
    //the rest of class definition
}
```

The two definitions are equivalent.

Joanna Klukowska
joannakl@cs.nyu.edu

4.1.3 Constructors and inheritance

Constructors of the superclass are not inherited by its subclass. They can be invoked using the super keyword from within subclass constructors.

Example:

Given the following code

```
public class C1 {
    public C1 ( ) { //do something }
    public C1 ( int c ) { //do something }
    //class definition
}
public class C2 extends C1 {
    public C2 ( int c, int x ) {
        super(c);
        //do something
    }
    //class definition
}
```

an object of the C2 class can be used using two parameter constructor, BUT NOT one parameter constructor or the default constructor.

```
C2 object1 = new C2 ( 3, 5 );  //is valid
C2 object2 = new C2 ( 3);      //is NOT valid
C2 object3 = new C2 ();        //is NOT valid
```

Constructor chaining: constructing an instance of a class invokes the constructors of ALL the superclasses along the inheritance chain.

A constructor may

- invoke an overloaded constructor (of its own class), or
- invoke its superclass constructor (this has to be done in the first line of the constructor).

If neither of these happens explicitly, the Java compiler automatically adds super() as the first statement in the constructor. (This may cause problems if the superclass does not have a default constructor.)

Joanna Klukowska
joannakl@cs.nyu.edu

4.1.4 Overriding methods of the superclass

Overriding is redefining a method of a superclass in a subclass. The **overridden method** has to have the same name, parameter list and return type as the method in the superclass.

Example:

```
public class C1 {
    public void sayHi() {
        System.out.println("C1 says hi");
    }
    //rest of class definition
}
public class C2 extends C1 {
    @Override
    public void sayHi() {
        super.sayHi();
        System.out.println("C2 says hi");
    }
    //rest of class definition
}
```

The class `C2` overrides the inherited method `sayHi()` and uses the `super` keyword to access the overridden method (in this case the keyword `super` is not optional).

- An overridden method of a superclass can be accessed using the `super` keyword. (This only works over one level of inheritance, i.e., there is no `super.super._____`.
- Static methods of the superclass are not overridden. If a subclass defines a static method whose name matches the name of the static method in the superclass, one needs to use the name of the superclass, not the keyword `super` to access it.

  ```
  SuperClassName.staticMethodName(...);
  ```

  not, `super.staticMethodName();`.
- Use the override annotation, `@Override`, when declaring methods that override methods of the superclass. This way the Java compiler double checks that your method truly overrides the method of the superclass.

Overriding vs. overloading

If the method in a subclass does not match the method of its superclass exactly in its parameter list, then it only overloads the method of the superclass (both still can be accessed).

Joanna Klukowska
joannakl@cs.nyu.edu

Example:

```
public class Test {
  public static void main( String [] args ) {
    C2 c = new C2();
    c.foo( 10 );
    c.foo( 10.0 );
  }
}
class C1 {
  public void foo (double x ) {
    System.out.println("C1.foo() called " );
  }
}
class C2 extends C1{
  //this is overriding
  public void foo (double x ) {
    System.out.println("C2.foo() called " );
  }
}
```

Output:

```
C2.foo() called
C2.foo() called
```

```
public class Test {
  public static void main( String [] args ) {
    C2 c = new C2();
    c.foo( 10 );
    c.foo( 10.0 );
  }
}
class C1 {
  public void foo (double x ) {
    System.out.println("C1.foo() called " );
  }
}
class C2 extends C1{
  //this is overloading
  public void foo (int x ) {
    System.out.println("C2.foo() called " );
  }
}
```

Output:

```
C2.foo() called
C1.foo() called
```

4.2 `Object` class and its methods

Every class in Java inherits automatically from the `Object` class (you do not need to do anything in order for your class to inherit from `Object`).

The Object class has several methods that every other class inherits. Most of the times, unless the subclass overrides these methods they are not very interesting. See the documentation for the Object class at

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

`toString()` method that we have been using for some time now, is inherited from the `Object` class. Whenever you write your own version of `toString()` method, you are overriding the one in the `Object` class.

4.3 Polymorphism

Polymorphism - a variable of a superclass / supertype / base type can refer to a variable of a subclass / subtype / derived type.

Dynamic binding - a method can be implemented in several classes along the inheritance chain. JVM decides which method should run at runtime.

A variable has two (possibly different) types associated with it:

- **declared type** - the type listed in variable declaration,
- **actual type** - the type of object that variable references.

The method that is invoked by a variable at runtime is determined by its actual type, but the compiler can only determine appropriateness of method calls based on the declared type.

4.3.1 Casting and `instaceof` operator

```
Person p = new Student ( ...  );
```

is reffered to as **implicit casting**. It is done automatically (you do not need to tell the compiler to do it) because an instance of the class `Student` is an instance of the class `Person` (student is-a person).

On the other hand, the statements

```
Person p = new Student ( ...  );
Student s = p;
```

will cause a compiler error because the compiler only uses the declared type of variable `p` and since person is not a student it cannot perform the assignment. In this particular case `p` references a `Student` object, but it might not always be the case (and the compiler certainly cannot know that). The way around it is explicit casting:

```
Person p = new Student ( ...  );
Student s = (Student) p;
```

The `instanceof` operator allows you to make sure that the particular variable references the object that is of a particular type.

```
refVariable instanceof ClassName
```

evaluates to true if `refVariable` references an instance of class `ClassName`, and `false` otherwise.

4.4 Code Example

Source code: Race02.java, RaceContender02.java, Rabbit02.java, Duck02.java, Snail02.java

This program is the second race simulation. It demonstrates several concepts:

Inheritance: It uses the class RaceContender02 as a base class for different kinds of actual contenders: Rabbit class, Duck class and Snail class.

Not-inheriting constructors: Each of the three classes defines its own constructor.

Use of `super` in constructors: The constructors of subclasses make a call to the superclass' constructor.

Method overriding: Each of the three classes defines its own move method that implements moves more characteristic for the class rather than using the generic move() method implemented in RaceContender02 class.

Polymorphism: The winner reference variable has declared type of RaceContender02, but when the winner is determined it refers to an object of type Rabbit, Duck, Snail or RaceContender02.

5 Abstract Classes and Interfaces

This section mentions very basic ideas of abstract classes and interfaces. We will see many more details and applications throughout the semester.

Joanna Klukowska
joannakl@cs.nyu.edu

5.1 Abstract Classes

An **abstract class** contains **abstract methods** (think of these as method-placeholders) that are implemented by concrete subclasses.

Example: Every `GeometricObject` should provide `getArea()` and `getPerimeter()` methods, but unless we know what actual shape it is, these methods cannot be implemented.

Syntax:

```
public/protected abstract ClassName {
    ...
    abstract returnType methodName () ;
    ...
}
```

Reasons for abstract classes:

- provide base/superclass class that guarantees that all subclasses provide certain methods (subclasses have to implement abstract methods of its superclass);
- ability to write more "generic" code

5.1.1 Code Example

Source code: Race03.java, RaceContender03.java, Rabbit03.java, Duck03.java, Snail03.java

This program is the third race simulation. It demonstrates several concepts:

Abstract class: RaceContender03 is now an abstract class with an abstract move() method. The three animal classes provide their own implementations of the method. RaceContender03 class cannot be instantiated (i.e. we cannot have an object of type RaceContender03 created using new operator).

Polymorphism: This time we have an array of racers. The array is of type RaceContender03 and contains references of type RaceContender03. But the elements that these references point to are of types Rabbit, Duck or Snail.

5.2 Interfaces

An interface is a class-like construct that contains only constants and abstract methods.

Syntax:

```
interface Name {
    ...
}
```

Note that all data fields have to be `public static final` and all methods have to be `public abstract` (since there is no choice about it, you can omit the access modifiers in definitions of interfaces).

A class can implement multiple interfaces.

Joanna Klukowska
joannakl@cs.nyu.edu

5.2.1 `Comparable` interface provided by Java

A class that implements `Comparable` interface has to provide `compareTo()` method. That is the only requirement of the `Comparable` interface.

5.2.2 Code Example

Source code: Race04.java, RaceContender04.java, Rabbit04.java, Duck04.java, Snail04.java

This program is the fourth (and final for these notes) race simulation. It demonstrates several concepts:

Comparable objects: RaceContender04 is still an abstract class with an abstract move() method. It also implements Comparable interface and provides the compareTo() method: the larger the value of position, the smaller the object (it is closest to the FINISH_LINE).

Finding smallest: At each iteration of the race we display the number of the race contender who is in the lead. This uses the idea of finding the smallest element in the array that is determined by compareTo() method.

Sorting an array of objects: The Arrays class provides sort() method that can be used with any class that implements Comparable interface. At the end of the race we display the list of all the contenders in the order determined by how close they got to the FINISH_LINE.