

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

Reading materials

Goodrich, Tamassia, Goldwasser (6th), chapters 1 and 2

Liang (10th): chapters 9,10,11,12,13

Oracle tutorial: Object oriented programming concepts, <https://docs.oracle.com/javase/tutorial/java/concepts/index.html>

Oracle tutorial: Classes and objects, <https://docs.oracle.com/javase/tutorial/java/javaOO/index.html>

Oracle tutorial: Inheritance and Polymorphism, <https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html>

Oracle tutorial: Interfaces, <https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html>

Oracle tutorial: What is a Path (and other file system facts)? <http://docs.oracle.com/javase/tutorial/essential/io/path.html>

Contents

1 Introduction to Data Structures Course	3
1.1 What are data structures?	3
1.2 What are algorithms?	3
1.3 The Phases of Software Development (Software Engineering)	4
2 Method Specification/Documentation	5
2.1 Javadoc	6
3 Object Oriented Design (Review)	7
3.1 Reference Variables vs. Objects	8
3.2 Object Composition (HAS-A relationship)	8
3.3 Class Design Guidance	9
4 Inheritance and Polymorphism	10
4.1 Inheritance (IS-A relationship)	10
4.1.1 super keyword	11
4.1.2 Superclass methods and data fields	11
4.1.3 Constructors and inheritance	12
4.1.4 Overriding methods of the superclass	13
4.2 Object class and its methods	14
4.3 Polymorphism	14



4.3.1 Casting and instanceof operator	15
5 Abstract Classes and Interfaces	15
5.1 Abstract Classes	15
5.2 Interfaces	16
5.2.1 Comparable interface provided by Java	16
6 Command-Line Arguments	16
7 File I/O	18
7.1 File Class	19
7.2 Reading Data from an Input File	19
7.3 Writing Data to Output Files	20



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 in a memory of a running program that are 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:

- **list**
- **stack**
- **queue**
- **tree**
- **binary tree**
- **hash table**

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.



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?

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:

```
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 (i.e., inline comments and documentation of private methods).
- 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 a 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` program 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.

3 Object Oriented Design (Review)

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

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).

3.1 Reference Variables vs. 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 = new Circle(5);  
Circle c2 = new 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?

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

Based on *Introduction to Java Programming* by Y. Daniel Liang.

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.

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.

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.

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).

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.

```
1 public class C1 {
2     protected int x;
3     public void setX ( int x ) {
4         this.x = x;
5     }
6     //the rest of class definition
7 }
8

1 public class C2 extends C1 {
2     protected float c;
3     public C2 ( float c, int x ) {
4         this.c = c;
5         setX(x);
6     }
7     //the rest of class definition
8 }
```

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 indicate 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

```
1 public class C2 extends C1 {
2     protected float c;
3     public C2 ( float c, int x ) {
4         this.c = c;
5         super.setX(x);
6     }
7     //the rest of class definition
8 }
```

The two definitions are equivalent.

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.

Given the following code

```
1 public class C1 {
2     public C1 ( ) { //do something }
3     public C1 ( int c ) { //do something }
4     //class definition
5 }
6
7

1 public class C2 extends C1 {
2     public C2 ( int c, int x ) {
3         super(c);
4         //do something
5     }
6     //class definition
7 }
```

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.)

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.

```
1 public class C1 {
2     public void sayHi() {
3         System.out.println("C1 says hi");
4     }
5     //rest of class definition
6 }
7
8
9
1 public class C2 extends C1 {
2     @Override
3     public void sayHi() {
4         super.sayHi();
5         System.out.println("C2 says hi");
6     }
7     //rest of class definition
8 }
```

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 (they are hidden). 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).

Example:

```
1 public class Test {
2     public static void main( String [] args ) {
3         C2 c = new C2 ();
4         c.foo( 10 );
5         c.foo( 10.0 );
6     }
7 }
8
9 class C1 {
10    public void foo (double x ) {
11        System.out.println("C1.foo() called " );
12    }
13 }
14
15 class C2 extends C1{
16     //this is overriding
17     public void foo (double x ) {
18         System.out.println("C2.foo() called " );
19     }
20 }
```

```
1 public class Test {
2     public static void main( String [] args ) {
3         C2 c = new C2 ();
4         c.foo( 10 );
5         c.foo( 10.0 );
6     }
7 }
8
9 class C1 {
10    public void foo (double x ) {
11        System.out.println("C1.foo() called " );
12    }
13 }
14
15 class C2 extends C1{
16     //this is overloading
17     public void foo (int x ) {
18         System.out.println("C2.foo() called " );
19     }
20 }
```

Output:

```
C2.foo() called
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, for example, 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 instanceof operator

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

is referred 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.

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.

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

5.2 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.

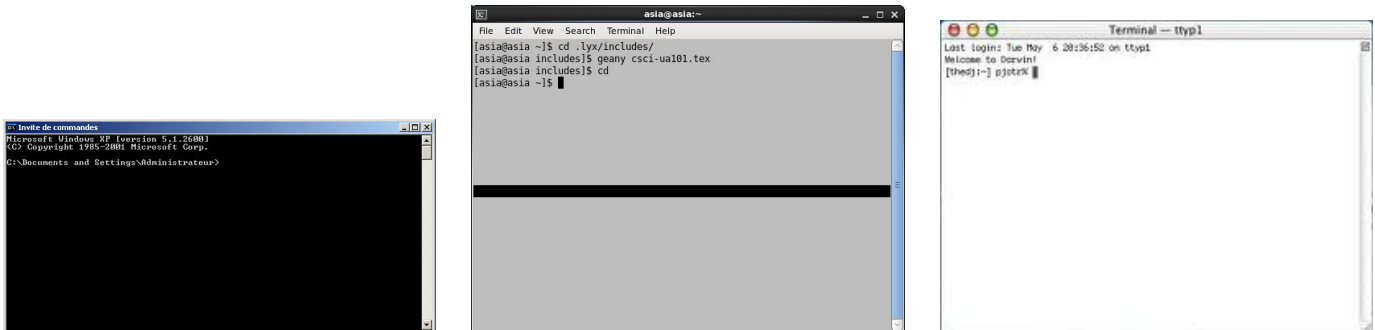
5.2.1 Comparable interface provided by Java

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

6 Command-Line Arguments

See <https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html>

What is a terminal or command-line? Many programs can be started from a terminal window (a.k.a., from a command line). Chances are that you never run a program that way. Here are images of a few different terminal environments:



In an environment like that, you run Java programs by specifying first the **java** command (this starts the Java Runtime Environment) followed by the name of the class that you are trying to run, followed by (optional) command line arguments. For example:

```
java Echo  
java Echo these are command line arguments
```

Comman-Line Arguments in Eclipse You are probably not going to start working in a terminal window right away (although you should definitely know how to do it!!!). Fortunately Eclipse (and other IDEs) has a way of specifying the command line arguments througih its graphical interface. This allows you to test the program that you are working on that might be used in a terminal-environment by someone else. Go to *Run* → *Run Configurations*. On the left hand side make sure that the class that you are working with is highlighted. Switch to *Arguments* tab and add your command line arguments in the *Program Arguments* text area. For the above **Echo** program I would just enter **these are command line arguments** (note that these are just arguments, the Eclipse environment takes care of calling **java** program that loads our **Echo** class).

The args array The Java Runtime Environment looks at the strings that follow the name of the class that it executes, packages those strings into an array of strings and then hands that array to the program's **main** function. The syntax for the **main** function in Java is

```
public static void main (String [] args)
```

The **args** array contains all of the strings that were specified by the user on the command line right after the name of the class that is now running. If the program is run as in the first line above, then the **args** array is empty (since there are no words specified after the name of the class). If the program is run as in the second line above, then the **args** array contains five strings: ["**these**", "**are**", "**command**", "**line**", "**arguments**"] and you can access that array as any other array: **args[0]**, **args[3]**, ..., **args.length**,

Let's look at a very simple class that uses the command line arguments. It echoes the arguments back to the console (not much, but gives you the idea of how to access the array).

```
1
2 public class EchoWords {
3     //The command line arguments are passed to your program inside
4     //the args array.
5     public static void main(String[] args) {
6
7         //First check how many command line arguments
8         //were provided
9         System.out.printf("There are %d command line arguments.\n",
10             args.length );
11
12         //Then print all the arguments to the console one per line
13         for (int i = 0; i < args.length; i++ ) {
14             System.out.println(args[i]);
15         }
16     }
17 }
```

Spaces in Command Line Arguments Sometimes the command line argument requires a space within it. It happens often when you are trying to specify a name of the input file and its path contains a space in one or more names of the directories. You can let the JRE that this is really one argument by surrounding it with double quotes. Rerun the above program specifying

```
java Echo "these are" "command line" arguments
```

(or in Eclipse "**these are**" "**command line**" **arguments**). You should see only three command line arguments, but some of them have more than one word.

Why use command line arguments? Many programs need extra parameters or configuration options when they are started. These things can be passed to such programs from the command line. Many programs are not really interactive (not designed to be run by users who are going to sit there and respond of questions/prompts), but may need some initial info about the data sources, locations of log files, etc. These things can be passed to such programs from the command line. Other programs need input data files (like some of your projects) and it is nice to be able to specify them upfront when the application starts, rather than have the user type the name after the prompt.

7 File I/O

The programs that you write rarely interact with just a human user. Most useful programs log errors / events in log file, read input data from databases (or other input files), write their results to output files.

In order to work with files, you need to be familiar with a few concepts about **file systems**, i.e.,

- how are the files and directories organized on your computer
- how a program use the **path** to a file
- what the difference is between an **absolute** and a **relative** path
- permissions associated with files (can we read it, write to it, execute it as if it was a program)
- how are sizes of files calculated and represented
- ...

See <http://docs.oracle.com/javase/tutorial/essential/io/path.html> for a brief description of these concepts if you are not familiar with those.

7.1 File Class

In Java the **File** class provides an abstraction for a file or a directory stored on disk of your computer (in fact, you can create a **File** object for a file that does not even exist).

File class documentation: <http://docs.oracle.com/javase/8/docs/api/java/io/File.html>.

Using methods of this class we can determine if a particular file exists, how big it is, when it was modified. We can even change properties of files stored on a hard-disk of a computer using this class.

Code Example See `TestFileClass.java`

The `TestFileClass.java` has a **hard-coded filename** in the program. This does not make it very portable: every time you want to learn about a different file you have to modify the program, recompile it and run it again. Hard-coding refers to a practice (a bad practice) of writing into the code information that may need to frequently change. Avoid writing programs with hard-coded values - they are not portable and will rarely work for anybody else.

Challenge: Modify this program to allow the user to specify a path to a file as a command line argument.

7.2 Reading Data from an Input File

The same **Scanner** class that we used for reading data from the console/standard input can be used to read data from a file.

Scanner class documentation <http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>.

To create a **Scanner** object we need to use a constructor that takes a **File** object as a parameter:

```
Scanner ( File filename );
```

This constructor may throw an instance of **IOException** if, for whatever reason, it cannot create the connection between the file on disk and the object. To avoid that, we should use methods from the **File** class to verify if the file exists and if the user running the program has permissions to access it. If the **File** class tells us that the file does not exist, there is no reason to call the **Scanner** constructor - it will fail. Even after all the checks, there is still a tiny chance that the constructor might fail. To make sure that our programs do not crash with nasty exception messages, we need to surround a call to the constructor with a try catch block that allows us to gently terminate the program or ask the user for corrections.

We can use all the same methods for reading data from files as we did for reading data from the user (standard input).

When working with files, you should always close a **Scanner** object (i.e., call the **close()** method) when you are done working with this. It avoids possible file corruption and lets the rest of your system know that you are finished with the file.

Code Example See **ReadData.java** code.

Challenge: Once again, the above example has a hard-coded name of the file in it. Rewrite it to allow the user to specify the name of the file as a command line argument.

There are many other classes in Java that can be used to read input from text files and from binary files. In some cases, they provide a more efficient way of reading data. You are not required to know these classes for this course.

7.3 Writing Data to Output Files

The **PrintWriter** objects can be used to write data to output files (instead of writing it to the console, for example). It has the same methods that we used for writing to the console: **print()**, **println()** and **printf()**.

PrintWriter class documentation: <http://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html>

To create a **PrintWriter** object we need to use a constructor that takes a **File** object as a parameter:

```
PrintWriter(File file)
```

Similarly to the **Scanner** constructor, this constructor may throw an instance of **IOException** and we should use the methods of the **File** class to verify if the program can actually write to the given output file.

If the **File** object represents a file on disk that does not exist yet, the execution of the **PrintWriter** constructor will create that file on disk.

WARNING: If the **File** object represents a file on disk that does exist, the execution of the **PrintWriter** constructor will **OVERWRITE** the existing file (and there is no way to recover the old content).