Walk through 4 - Java Tutorial

Topic User Data Types

1. Introduction

In the first lab you wrote and run your first Java program (Hello World) using Eclipse. In the second and third labs you learned how to deal with:

- Primitive data types in Java
- Conditions and Loops
- Arrays
- Static methods in Java
- Recursion in Java

The Java programs that you have so far developed were all using static methods defined in Java classes. However, in object-oriented programming, we write Java code to create new data types, specifying the values and operations to manipulate those values. The idea originates from modeling (in software) real-world entities such as electrons, people, buildings, or solar systems and extends readily to modeling abstract entities such as bits, numbers, programs or operating systems.

1. Lab Objectives

The objective of this lab is to learn:

- Data type (String, Color)
- **Properties of reference types.**
- Class and objects
- Basic elements of a data type (API, Access modifiers, Instance variables, Constructors, instance methods, Test client)
- Creating and Using objects (invoking instance methods, objects as arguments, objects as return values, arrays of objects)
- Interface inheritance
- Class Hierarchy (implementation inheritance)

2. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer.
2. Completed Lab walk-through 1-3 from the previous weeks.
3. Read chapter 1.2 in your textbook.
4. **Before you start doing this lab, in your Eclipse workspace create a project named 2XB3_Lab4 inside a package named cas.2XB3.lab4.wt. Add classes to this project as you walk through this instruction.**

Note: YOU NEED TO HAVE YOUR OWN STORAGE DEVICE (E.G., A USB KEY) IN ORDER TO CREATE THE ECLIPSE WORKSPACE.

IT IS YOUR RESPONSIBILITY TO KEEP YOUR STORAGE DEVICE SAFE (WITH NECESSARY BACKUPS) FOR FUTURE USE OF THE PROJECTS YOU ARE CREATING IN THE LABS.

3. Lab Exercise

3.1 Using Data Types

A *data type* is a set of values and a set of operations defined on those values. The primitive data types that you have been using are supplemented in Java by extensive libraries of *reference types* that are tailored for a large variety of applications. In this section, we consider reference types for string processing and image processing.

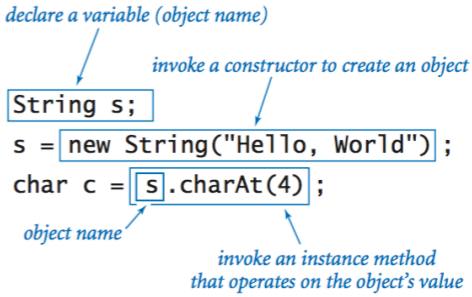
Strings.

You have already been using a data type that is not primitive—the `String` data type, whose values are sequences of characters. We specify the behavior of a data type in an *application programming interface* (API). Here is a partial API for Java's String data type:

`public class String`

	`String(String s)`	*create a string with the same value as* `s`
`int`	`length()`	*number of characters*
`char`	`charAt(int i)`	*the character at index* `i`
`String`	`substring(int i, int j)`	*characters at indices* `i` *through* `(j-1)`
`boolean`	`contains(String substring)`	*does this string contain* `substring` *?*
`boolean`	`startsWith(String pre)`	*does this string start with* `pre` *?*
`boolean`	`endsWith(String post)`	*does this string end with* `post` *?*
`int`	`indexOf(String pattern)`	*index of first occurrence of* `pattern`
`int`	`indexOf(String pattern, int i)`	*index of first occurrence of* `pattern` *after* `i`
`String`	`concat(String t)`	*this string with* `t` *appended*
`int`	`compareTo(String t)`	*string comparison*
`String`	`toLowerCase()`	*this string, with lowercase letters*
`String`	`toUpperCase()`	*this string, with uppercase letters*
`String`	`replaceAll(String a, String b)`	*this string, with* `a`*s replaced by* `b`*s*
`String[]`	`split(String delimiter)`	*strings between occurrences of* `delimiter`
`boolean`	`equals(Object t)`	*is this string's value the same as* `t`*'s ?*
`int`	`hashCode()`	*an integer hash code*

The first entry, with the same name as the class and no return type, defines a special method known as a *constructor*. The other entries define *instance methods* that can take arguments and return values.



- *Declaring variables.* You declare variables of a reference type in precisely the same way that you declare variables of a primitive type. A declaration statement does not create anything; it just says that we will use the variable name `s` to refer to a `String` object.
- *Creating objects.* Each data-type value is stored in an *object*. When a client invokes a constructor, the Java system creates (or *instantiates*) an individual object (or *instance*). To invoke a constructor, use the keyword `new`; followed by the class name; followed by the constructor's arguments, enclosed in parentheses and separated by commas.
- *Invoking instance methods.* The most important difference between a variable of a reference type and a variable of a primitive type is that you can use reference-type variables to invoke the *instance methods* that implement data-type operations (in contrast to the built-in syntax involving operators such as `+`* that we used with primitive types).

Now, we consider various string-processing examples.

- *Data-type operations.* The following examples illustrate various operations for the `String` data type.

```
String a = new String("now is");
String b = new String("the time");
String c = new String(" the");
```

instance method call	*return type*	*return value*
`a.length()`	`int`	`6`
`a.charAt(4)`	`char`	`'i'`
`a.substring(2, 5)`	`String`	`"w i"`
`b.startsWith("the")`	`boolean`	`true`
`a.indexOf("is")`	`int`	`4`
`a.concat(c)`	`String`	`"now is the"`
`b.replace("t", "T")`	`String`	`"The Time"`
`a.split(" ")`	`String[]`	`{ "now", "is" }`
`b.equals(c)`	`boolean`	`false`

- *Code fragments.* The following code fragments illustrate the use of various string-processing methods.

extract file name and extension from a command-line argument

```
String s = args[0];
int dot = s.indexOf(".");
String base      = s.substring(0, dot);
String extension = s.substring(dot + 1, s.length());
```

print all lines on standard input that contain a string specified as a command-line argument

```
String query = args[0];
while (StdIn.hasNextLine())
{
   String line = StdIn.readLine();
   if (line.contains(query))
      StdOut.println(line);
}
```

is the string a palindrome?

```
public static boolean isPalindrome(String s)
{
   int n = s.length();
   for (int i = 0; i < n/2; i++)
      if (s.charAt(i) != s.charAt(n-1-i))
         return false;
   return true;
}
```

translate from DNA to mRNA (replace 'T' with 'U')

```
public static String translate(String dna)
{
   dna = dna.toUpperCase();
   String rna = dna.replaceAll("T", "U");
   return rna;
}
```

Task 1: Write a function to check the validity of a DNA sequence.

```
/*************************************************************************
 *  Compilation:  javac PotentialGene.java
 *  Execution:    java PotentialGene < input.txt
 *
 *  Determines whether a a DNA string corresponds to a potential gene
 *      - length is a multiple of 3
 *      - starts with the start codon (ATG)
 *      - ends with a stop codon (TAA or TAG or TGA)
 *      - has no intervening stop codons (i.e. a stop codon cannot be in the
 *     middle of the string.
 *
 *  % java PotentialGene ATGCGCCTGCGTCTGTACTAG
 *  true
 *
 *  % java PotentialGene ATGCGCTGCGTCTGTACTAG
 *  false
 *
*************************************************************************/

public class PotentialGene {

    public static boolean isPotentialGene(String dna) {

        // Your code here
        // Length is a multiple of 3.
```

```
        // Starts with start codon.
        // No intervening stop codons.
        // Ends with a stop codon.

        return false;
    }

    public static void main(String[] args) {
        String dna = args[0];
        StdOut.println(isPotentialGene(dna));
    }
}
```

3.2 Creating Data Types

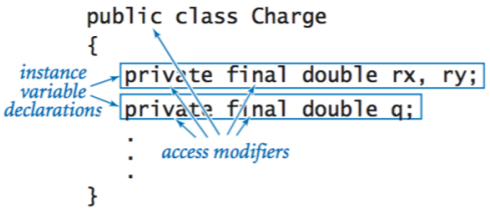
In this section, we introduce the Java mechanism that enables us to create user-defined data types.

Basic elements of a data type.

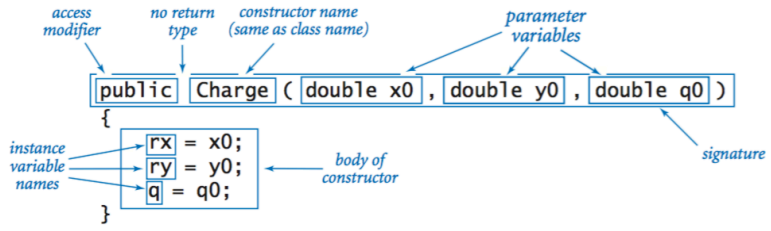
- *API.* The application programming interface is the contract with all clients and, therefore, the starting point for any implementation.

public class Charge		
	Charge(double x0, double y0, double q0)	
double	potentialAt(double x, double y)	*electric potential at (x, y) due to charge*
String	toString()	*string representation*

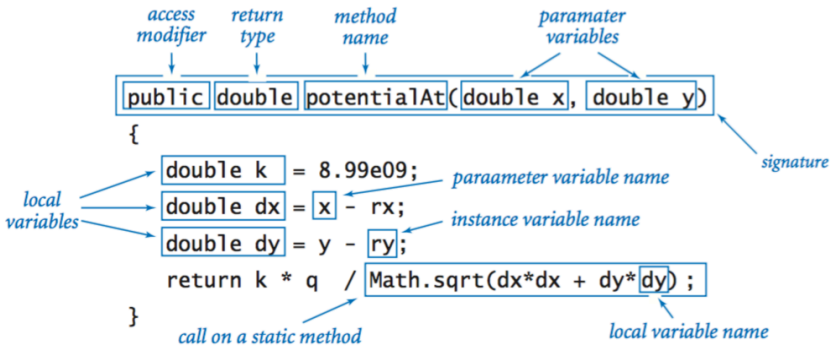
- *Class.* In Java, you implement a data type in a `class`. As usual, we put the code for a data type in a file with the same name as the class, followed by the `.java` extension.
- *Access modifiers.* We designate every instance variable and method within a class as either `public` (this entity is accessible by clients) or `private` (this entity is not accessible by clients). The `final` modifier indicates that the value of the variable will not change once it is initialized—its access is read-only.
- *Instance variables.* We declare *instance variables* to represent the data-type values in the same way as we declare local variables, except that these declarations appear as the first statements in the class, not inside `main()` or any other method.



- *Constructors.* A constructor is a special method that creates an object and provides a reference to that object. Java automatically invokes a constructor when a client program uses the keyword `new`. Each time that a client invokes a constructor, Java automatically
  - Allocates memory for the object
  - Invokes the constructor code to initialize the instance variables
  - Returns a reference to the newly created object



- *Instance methods.* To implement instance methods, we write code that is precisely like code for implementing static methods. The one critical distinction is that instance methods can perform operations on instance variables.



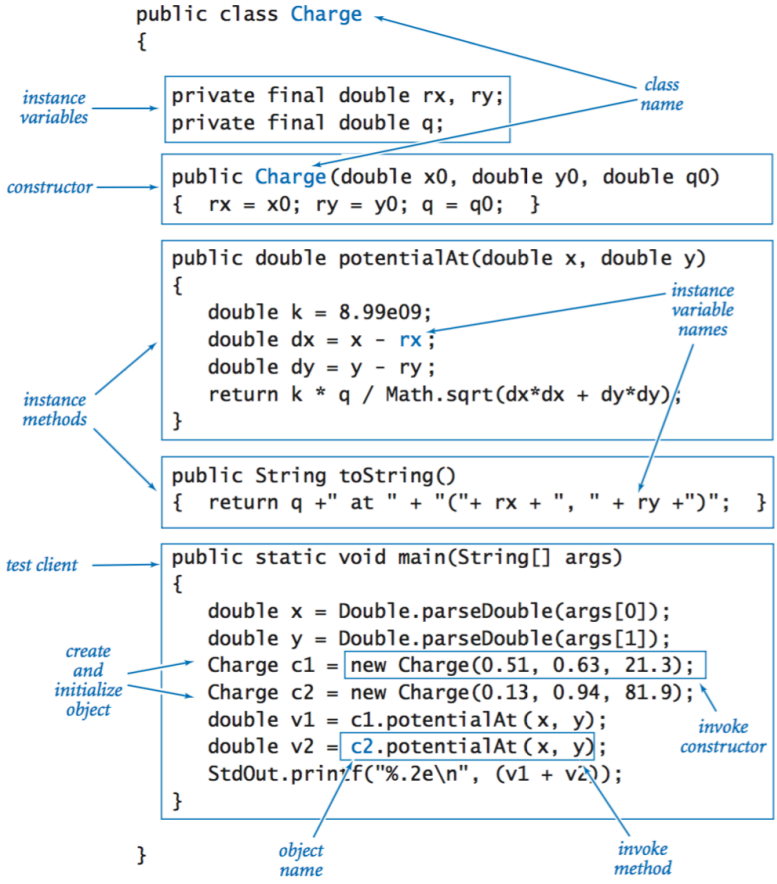
- *Variables within methods.* The Java code we write to implement instance methods uses three kinds of variables. Parameter variables and local variables are familiar. Instance variables are completely different: they hold data-type values for objects in a class.

variable	*purpose*	*example*	*scope*
instance	to specify data-type value	`rx, ry`	class
parameter	to pass value from client to method	`x, y`	method
local	for temporary use within method	`dx, dy`	block

Each object in the class has a value: the code in an instance method refers to the value for the object that was used to invoke the method.

- *Test client.* Each class can define its own `main()` method, which we reserve for unit testing. At a minimum, the test client should call every constructor and instance method in the class.

In summary, to define a data type in a Java class, you need instance variables, constructors, instance methods, and a test client.



Stopwatch.

Stopwatch.java implements the following API:

public class Stopwatch	
Stopwatch()	*create a new stopwatch and start it running*
double elapsedTime()	*return the elapsed time since creation, in seconds*

It is a stripped-down version of an old-fashioned stopwatch. When you create one, it starts running, and you can ask it how long it has been running by invoking the method `elapsedTime()`.

Task 2: Implement the Stopwatch API

```
/*************************************************************************
 *  Compilation:  javac Stopwatch.java
```

```
 *  Execution:    java Stopwatch n
 *  Dependencies: none
 *
 *  A utility class to measure the running time (wall clock) of a program.
 *
 *  The {@code Stopwatch} data type is for measuring
 *  the time that elapses between the start and end of a
 *  programming task (wall-clock time).
 *
 */

public class Stopwatch {

    private final long start;

    /**
     * Initializes a new stopwatch.
     */
    public Stopwatch() {
        start = System.currentTimeMillis();
    }

    /**
     * Returns the elapsed CPU time (in seconds) since the stopwatch was
created.
     *
     * @return elapsed CPU time (in seconds) since the stopwatch was created
     */
    public double elapsedTime() {
     // your code here
        return (now - start) / 1000.0;
    }


    /**
     * Unit tests the {@code Stopwatch} data type.
     * Takes a command-line argument {@code n} and computes the
     * sum of the square roots of the first {@code n} positive integers,
     * using {@code Math.sqrt()}.
     * It prints to standard output the sum and the amount of time to
     * compute the sum. Note that the discrete sum can be approximated by
     * an integral - the sum should be approximately 2/3 * (n^(3/2) - 1).
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
```

```
        // sum of square roots of integers from 1 to n using Math.sqrt(x).
        Stopwatch timer1 = new Stopwatch();
        double sum1 = 0.0;
        for (int i = 1; i <= n; i++) {
 //your code here
        }
        double time1 = timer1.elapsedTime();
        StdOut.printf("%e (%.2f seconds)\n", sum1, time1);
    }
}
```

3.3 Designing Data Types

In this section we discuss *encapsulation*, *immutability*, and *inheritance*, with particular attention to the use of these mechanisms in *data-type design* to enable modular programming, facilitate debugging, and write clear and correct code.

Encapsulation.

The process of separating clients from implementations by hiding information is known as *encapsulation*. We use encapsulation to enable modular programming, facilitate debugging, and clarify program code.

- *Private.* When you declare an instance variable (or method) to be `private`, you are making it impossible for any client (code in another class) to directly access that instance variable (or method). This helps enforce encapsulation.
- *Limiting the potential for error.* Encapsulation also helps programmers ensure that their code operates as intended. To understand the problem, consider Counter.java, which encapsulates a single integer and ensures that the only operation that can be performed on the integer is *increment by 1*.

public class Counter	
Counter(String id, int max)	*create a counter, initialized to 0*
void increment()	*increment the counter unless its value is max*
int value()	*return the value of the counter*
String toString()	*string representation*

Without the `private` modifier, a client could write code like the following:

```
Counter counter = new Counter("Volusia");
counter.count = -16022;
```

With the `private` modifier, code like this will not compile.

Immutability.

immutable	*mutable*
String	Turtle
Charge	Picture
Color	Histogram
Complex	StockAccount
Vector	Counter
	Java arrays

An object from a data type is *immutable* if its data-type value cannot change once created. An *immutable data type* is one in which all objects of that type are immutable.

- *Advantages of immutability.* We can use immutable objects in assignment statements (or as arguments and return values from methods) without having to worry about their values changing. This makes immutable type easier to reason about and debug.
- *Cost of immutability.* The main drawback of immutability is that a new object must be created for every value.
- *Final.* When you declare an instance variable as `final`, you are promising to assign it a value only once. This helps enforce immutability.
- *Reference types.* The `final` access modifier does not guarantee immutability for instance variables of mutable types. In such cases, you must make a *defensive copy*.

Task 3: Create an immutable Point data type that implements the following API:

public class Point	
Point(double x, double y)	
double distanceTo(Point q)	*Euclidean distance between this point and q*
String toString()	*string representation*

```
/*************************************************************************
***
 *  Compilation:  javac Point.java
 *  Execution:    java Point
 *
 *  Immutable data type for 2D points.
 *
*************************************************************************
**/

public class Point {
    private double x;   // Cartesian
    private double y;   // coordinates

    // create and initialize a point with given (x, y)
    public Point(double x, double y) {
     //your code here
    }

    // return Euclidean distance between invoking point p and q
    // i.e. sqrt((this.x - that.x)^2 + (this.y - that.y)^2)
    public double distanceTo(Point that) {
     //your code here
```

```
    }

    // draw point using standard draw
    public void draw() {
        StdDraw.point(x, y);
    }

    // draw the line from the invoking point p to q using standard draw
    public void drawTo(Point that) {
        StdDraw.line(this.x, this.y, that.x, that.y);
    }

    // return string representation of this point
    public String toString() {
        //your code here
    }

    // test client
    public static void main(String[] args) {
        Point p = new Point(0.6, 0.2);
        StdOut.println("p  = " + p);
        Point q = new Point(0.5, 0.5);
        StdOut.println("q  = " + q);
        StdOut.println("dist(p, q) = " + p.distanceTo(q));
    }
}
```

Interface inheritance (subtyping).

Java provides the `interface` construct for declaring a relationship between otherwise unrelated classes, by specifying a common set of methods that each implementing class must include. Interfaces enable us to write client programs that can manipulate objects of varying types, by invoking common methods from the interface.

- *Defining an interface.* Function.java defines an interface for real-valued functions of a single variable.

```
public interface Function {
    public abstract double evaluate (double x);
}
```

  The body of the interface contains a list of *abstract methods*. An abstract method is a method that is declared but does not include any implementation code; it contains only the method signature. You must save a Java interface in a file whose name matches the name of the interface, with a `.java` extension.

- *Implementing an interface.* To write a class that implements an interface, you must do two things.

- Include an `implements` clause in the class declaration with the name of the interface.
- Implement each of the abstract methods in the interface.

For example, Square.java and GaussianPDF.java implements the `Function` interface.

- *Using an interface.* An interface is a reference type. So, you can declare the type of a variable to be the name of an interface. When you do so, any object you assign to that variable must be an instance of a class that implements the interface. For example, a variable of type `Function` may store an object of type `Square` or `GaussianPDF`.

```
Function f1 = new Square();
Function f2 = new GaussianPDF();
Function f3 = new Complex(1.0, 2.0);   // compile-time
error
```

When a variable of an interface type invokes a method declared in the interface, Java knows which method to call because it knows the type of the invoking object. This powerful programming mechanism is known as *polymorphism* or *dynamic dispatch*.

Implementation inheritance (subclassing).

Java also supports another inheritance mechanism known as *subclassing*. The idea is to define a new class (*subclass*, or *derived class*) that inherits instance variables (state) and instance methods (behavior) from another class (*superclass*, or *base class*), enabling code reuse. Typically, the subclass redefines or *overrides* some of the methods in the superclass.

Task 4. Submit your work

Once all tasks are completed, you should submit your Eclipse project. Follow the instructions below for submission:

- Include a .txt file named last_name_initials.txt in the root of the project containing on separate lines: Full name, student number, any design decisions/assumptions you feel need explanation or attention.
- After checking the accuracy and completeness of your project, save everything and right-click on the name of the project, select Export->General->Archive File.
- Ensure that just your project has a check-mark beside it, and select a path to export the project to. The filename of the zipped project must follow this format: *macID_Lab4.zip.* Check the option to save the file in .zip format. Click Finish to complete the export.
- Go to Avenue and upload your zipped project to 'Lab Walk-through 4 – Lab Section X)
- IMPORTANT: You MUST export the FULL Eclipse project. Individual files (e.g. java/class files) will NOT be accepted as a valid submission.

4. Practice Problems

1. Write a function `reverse()` that takes a string as an argument and returns a string that contains the same sequence of characters as the argument string but in reverse order.

2. Write a static method `isValidDNA()` that takes a string as its argument and returns `true` if and only if it is composed entirely of the characters `A`, `T`, `C`, and `G`.

3. Implement a data type Rational.java numbers that supports addition, subtraction, multiplication, and division.

public class Rational	
Rational(int numerator, int denominator)	
Rational plus(Rational b)	*sum of this number and b*
Rational minus(Rational b)	*difference of this number and b*
Rational times(Rational b)	*product of this number and b*
Rational divides(Rational b)	*quotient of this number and b*
String toString()	*string representation*

4. Create a `Rectangle` ADT that represents a rectangle. Represent a rectangle by two points. Include a constructor, a `toString` method, and a method for computing the area.

5. **Encapsulation.** Is the following class immutable?

```
import java.util.Date

public class Appointment {
    private Date date;
    private String contact;

    public Appointment(Date date) {
        this.date = date;
        this.contact = contact;
    }

    public Date getDate() {
        return date;
    }
```