CS 284: Data Structures
Set 1

Instructor: Igor Faynberg
Webpage:
http://www.cs.stevens.edu/~faynberg/Cloud/aboutme.htm
E-mail: ifaynber@stevens.edu

Week 1 Objective

- Introduction to Java

Koffman and Wolfgang Appendix A

Java is Object Oriented

Classes and Objects

- Class definitions in .java files
- Def: a *class* is a named description for a group of entities that have the same characteristics
- *Objects* or *instances* of the class is the group of entities
- The characteristics are the attributes (data fields) for each object and the operations (methods) that can be performed on these objects

Primitive Data Types

- byte -128 to 127
- short -32,768 to 32,767
- int -2,147,483,648 to 2,147,483,647
- long -9,223,372,036,854,775,808 to …
- float $\pm 10^{38}$ incl. 0 with 6 digits of precision
- double $\pm 10^{308}$ incl. 0 with 15 digits of precision
- char Unicode character set
- boolean true, false

Methods

- Method: a group of statements to perform a particular operation (called function in many other languages)
- Instance Methods: Applied to an object using dot notation
- `object.method(arguments)`
- E.g. the `println` method that can be applied to `PrintStream` object `System.Out`
  - `System.out.println("The value of x is "+x);`

Static Methods

- `static char minChar(char ch1, char ch2)`
- **static** indicates that it is a *static or class method*
  - There is one per class, not one per object like instance methods
- Called using dot notation
  - `char ch=ClassName.minChar('a','A');`
- Static methods cannot call instance methods
- Used for pure procedures (no global data involved)

The `main` Method

- Point where execution begins

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

public:

static:

void:

Defining Your Own Classes

- A Java program is a collection of classes
- For example:
  - A class Person may store:
    - Given name
    - Family name
    - ID number
    - Year of birth
  - It can perform operations such as:
    - Calculate person's age
    - Test whether two Person objects refer to same person
    - Determine if the person is old enough to vote
    - Get one or more of the data fields from the Person object
    - Set one or more of the data fields of the Person object

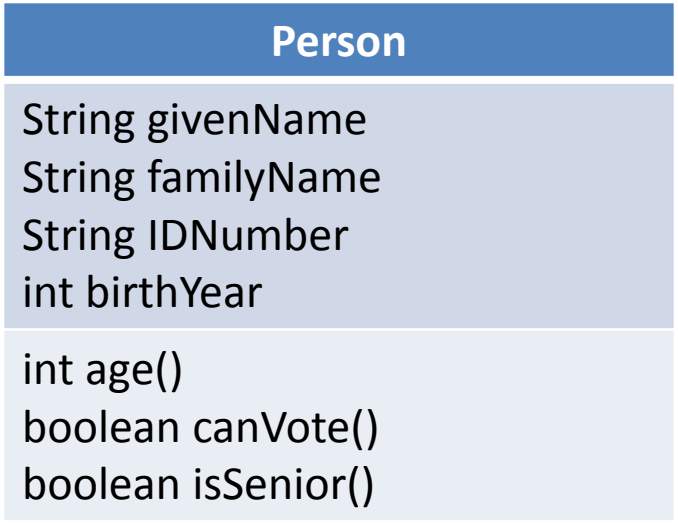
Rectangle Example

```
public class Rectangle{
	public double width;
	public double height;

	// constructor
	public Rectangle(double x, double y)
	{
		width = x;
		height = y;
	}

	public double area()
	{
		return width*height;
	}
}
```

Rectangle Example

```
// in main() method
// create a rectangle with width 3.5 and height 2.6
Rectangle rect = new Rectangle(3.5, 2.6);

// get its area
double ar;
ar = rect.area();
```

Universal Modeling Language (UML) Diagram (introduction)



Data fields (instance variables)

Methods

```
/** Person is a class that represents a human being.
 *  @author Koffman and Wolfgang
 * */

public class Person {
  // Data Fields
  /** The given name */
  private String givenName;

  /** The family name */
  private String familyName;

  /** The ID number */
  private String IDNumber;

  /** The birth year */
  private int birthYear = 1900;

  // Constants
  /** The age at which a person can vote */
  private static final int VOTE_AGE = 18;

  /** The age at which a person is considered a senior citizen */
  private static final int SENIOR_AGE = 65;
```

```
// Constructors
/** Construct a person with given values
    @param first The given name
    @param family The family name
    @param ID The ID number
    @param birth The birth year
 */
public Person(String first, String family, String ID, int birth)
{
  givenName = first;
  familyName = family;
  IDNumber = ID;
  birthYear = birth;
}

/** Construct a person with only an IDNumber specified.
    @param ID The ID number
 */
public Person(String ID)
 {
  IDNumber = ID;
}
```

```
 // Modifier Methods
 /** Sets the givenName field.
     @param given The given name
  */
 public void setGivenName(String given)
 {
   givenName = given;
 }

 /*  Sets the familyName field.
     @param family The family name
  */
 public void setFamilyName(String family)
 {
   familyName = family;
 }

 /** Sets the birthYear field.
     @param birthYear The year of birth
  */
 public void setBirthYear(int birthYear)
{
   this.birthYear = birthYear;
 }
```

```
// Accessor Methods
/** Gets the person's given name.
    @return the given name as a String
 */
public String getGivenName() {
  return givenName;
}

/** Gets the person's family name.
    @return the family name as a
 String
 */
public String getFamilyName()
{
  return familyName;
}

/** Gets the person's ID number.
    @return the ID number as a String
 */
public String getIDNumber()
{
  return IDNumber;
}
```

```
/** Gets the person's year of birth.
     @return the year of birth as an
int value
 */
public int getBirthYear()
{
  return birthYear;
}
```

```
 // Other Methods
 /**  Calculates a person's age at this year's birthday.
      @param year The current year
      @return the year minus the birth year
  */
 public int age(int year)
 {
   return year - birthYear;
 }

 /** Determines whether a person can vote.
     @param year The current year
     @return true if the person's age is greater than or
             equal to the voting age
  */
 public boolean canVote(int year)
{
   int theAge = age(year);
   return theAge >= VOTE_AGE;
 }
```

```
/** Determines whether a person is a senior citizen.
    @param year the current year
    @return true if person's age is greater than or
            equal to the age at which a person is
            considered to be a senior citizen
 */
public boolean isSenior(int year)
{
  return age(year) >= SENIOR_AGE;
}

/** Retrieves the information in a Person object.
    @return the object state as a string
 */
public String toString() {
  return "Given name: " + givenName + "\n"
      + "Family name: " + familyName + "\n"
      + "ID number: " + IDNumber + "\n"
      + "Year of birth: " + birthYear + "\n";
}
```

```
  /** Compares two Person objects for equality.
      @param per The second Person object
      @return true if the Person objects have same
              ID number; false if they don't
   */
  public boolean equals(Person per)
{
    if (per == null)
      return false;
    else
      return IDNumber.equals(per.IDNumber);
  }
}
```

Private Data Fields, Public Methods

- Provide better control of how data are accessed
- Allow the details of how data are stored and represented to be changed without affecting class's clients

Constructors

- We had two constructors for class *person:*
  - Four-parameter
  - One-parameter

- No-parameter constructor is not defined
- `Person p = new Person()` is invalid
- No-parameter constructor has to be explicitly defined if other constructors are defined

Use of `this.`

```
public void setBirthYear(int birthYear)
   {
     this.birthYear = birthYear;
   }
```

- `birthYear` is interpreted by the Java compiler as the local variable (parameter here) and not the data field with the same name

The Method `toString`

- To display the state of author1 (an instance of Person), we could use:
  - `System.out.println(author1.toString());`
  - `System.out.println(author1);`
- `System.out.println` and `System.out.print` automatically apply method `toString()` to an object that appears in their argument list

The Method `equals`

```
public boolean equals(Person per)
 {
    if (per == null)
      return false;
    else
      return IDNumber.equals(per.IDNumber);
  }
```

- We can look at `per`'s private ID number because `per` references an object of this class (Person)

testPerson

```
public class TestPerson {
  public static void main(String[] args) {
    Person p1 = new Person("Sam", "Jones", "1234", 1930);
    Person p2 = new Person("Jane", "Jones", "5678", 1990);
    System.out.println("Age of " + p1.getGivenName() +
                       " is " + p1.age(2012));
    if (p1.isSenior(2004))
      System.out.println(p1.getGivenName() +
                         " can ride the subway for free");
    else
      System.out.println(p1.getGivenName() +
                         " must pay to ride the subway");

    System.out.println("Age of " + p2.getGivenName() +
                       " is " + p2.age(2012));
    if (p2.canVote(2004))
      System.out.println(p2.getGivenName() + " can vote");
    else
      System.out.println(p2.getGivenName() + " can't vote");
  }
}
```

Arrays

Arrays of Arrays

```
double[][] matrix = new double[5][10];
```

- In Java, you can have two-dimensional arrays with rows of different sizes

```
char[][] letters = new char [5][];
letters[0] = new char[4];
letters[1] = new char[10];
```

Style

- Camel notation
  - `myVariable, thisLongIdentifier`
- Primitive type constants
  - all caps: `static final int MAX_SCORE=999`
- Postfix/prefix increment
  - `z=i++;`
  - `z=++i;`
  - Don't use `x*++i`

Type Compatibility and Conversion

- When mixed type operands are used, the type with the smaller range is converted to the type of the larger range
  - E.g. int+double is converted to double
  - *Widening* conversion
  -
```
int item = …;
double realItem = item; // valid ?
```
  -
```
double y = …;
int x=y; // valid ?
```

Referencing Objects

- `String greeting;`
- `greeting = "hello";`
  - String object "hello" is now referenced by `greeting`
  - `greeting` stores the address of where a particular String is stored.
- Primitive types store values not addresses
  - `x=3;`

References

- Two reference variables can reference the same object
  - `String welcome=greeting;`
  - copies the address in `greeting` to `welcome`
- Creating new objects
  - `String keyboard = new String("qwerty");`

Self-Check

- `String y=new String(“abc”);`
- `String z=“def”;`
- `String w=z;`

Control Statements

- if … else
- switch
- while
- do … while
- for

Examples

- Compute the sum of all even numbers from 2 to 200 unless they are multiples of 7
- Come up with a natural `do … while` example

Calling by Value

- In Java all arguments are call-by-value
  - If the argument is a primitive type, its value, not its address, are passed to the method
  - The method cannot modify the argument value and have this modification remain after returning
  - If the argument is of class type, it can be modified using its own methods and the changes are permanent
- Other languages also support call-by-reference

The `Math` Class

- Collection of useful methods
- All static

```
public class SquareRoots {
    public static void main(String[] args) {
        System.out.println("n \tsquare root");
        for (int n = 1; n <= 10; n++) {
            System.out.println(n + "\t" +
                               Math.sqrt(n));
        }
    }
}
```

The `String` Class

- Assume `keyboard` is a String that contains “string”

```
keyboard.charAt(0)
keyboard.length()
keyboard.indexOf('o')
keyboard.indexOf('y')
String upper=keyboard.toUpperCase();
```

Creates a new string object without changing `keyboard`

Strings are Immutable

- Strings are different from other objects in that they are immutable
  - A String object cannot be modified
  - New Strings are generated when changes are made

```
String myName = "Elliot Koffman";
myName = myName.substring(7) + ", " +
  myName.substring(0, 6);

myName[0]= 'X'; // invalid
myName.charAt(0)= 'X'; // invalid
```

Comparing Objects

```
String anyName = new String(myName);
anyName == myName  ?
```

- `==` operator compares the addresses and not the contents of the objects
- Use `equals, equalsIgnoreCase, compareTo, compareToIgnoreCase`
- Comparison methods need to be implemented for user-defined classes

Wrapper Classes for Primitive Types

- Primitive numeric types are not objects, but sometimes they need to be processed like objects
  - When?
- Java provides *wrapper classes* whose objects contain primitive-type values
  - `Float, Double, Integer, Boolean, Character`
  - They provide constructor methods to create new objects that “wrap” a specified value
  - Also provide methods to “unwrap”