Advanced Java

Youmin Lu

Department of Mathematics, Computer Science and Statistics

Bloomsburg University of Pennsylvania

Contents

Chapter One Object-Oriented Programming: Review 1

1.1 Members of a Class and Access Specifiers 1

1.2 Attributes and Constructors 2

1.3 Methods 4

1.4 Primitive vs. Reference Types 8

1.5 Inheritance and Polymorphism 10

1.6 Exceptions 13

1.7 Object Ordering, the Arrays and Collections Classes 13

Review Questions 16

Programming Projects 17

Chapter Two Container Classes 23

2.1 Lists and Iterators 23

2.2 Sets and Maps 28

2.3 Hashing Technique 30

Review Questions 32

Programming Projects 32

Chapter Three Generic Programming 37

3.1 Generic Methods 37

3.2 Generic Classes 38

Review Questions 41

Programming Projects 42

Chapter Four Recursive Programming 43

4.1 Basic Static Recursive Methods 43

4.2 Instance Recursive Methods 47

4.3 Recursion with Backtracking 50

Review Questions 58

Programming Projects 59

Chapter Five Multiple Threading 62

5.1 The Thread Class in Java Library 62

5.2 Extending the Thread Class 62

5.3 The Runnable Interface 64

5.4 The Interrupt Method and Interrupted Exception 66

5.5 Racing Conditions and Deadlocks 67

Review Questions 78

Programming Projects 79

Chapter Six Networking 81

6.1 The URL and URLConnection Classes 81

6.2 Clients and Server Using Sockets 84

6.3 Threaded Server 87

Review Questions 88

Programming Projects 89

Chapter Seven Java Database Connection (JDBC) 91

7.1 Relational Database and SQL 91

7.2 Connecting to a Database and Executing SQL Statements 94

7.3 Prepared Statements 97

7.4 Meta Data 98

7.5 Aggregate SQL Functions and Transactions 99

Review Questions 101

Programming Projects 102

Chapter Eight Servlets 103

8.1 HTML Basics 103

8.2 Cascading Style Sheet (CSS) Basics 112

8.3 CGI vs. Servlets 115

8.4 Defining and Activating a Servlet 116

8.5 Session Tracking 121

Review Questions 127

Programming Projects 128

Chapter Nine Java Server Pages (JSP) 130

9.1 Scripting Constructs and Comments 130

9.2 Predefined Variables 132

9.3 Directives and Exception Handling 134

9.4 JSP Actions 137

9.5 A Comprehensive Example 141

Review Questions 146

Programming Projects 147

Chapter Ten Extensible Markup Language (XML) 148

10.1 Structure of an XML Document 148

10.2 Parsing XML Document Using Java Programs 150

Review Questions 154

Programming Projects 155

References 156

Chapter One
Object-Oriented Programming: Review

Java is a totally object-oriented programming language. Object-oriented programming is to build programs using software objects which are created through classes. In object-oriented programming, a class is an extensible program-code-template for defining a group of instances. Each instance is called an object. Object-oriented programming changes the traditional separation of data and programs, and wraps data and procedures called methods into objects. The first advantage provided by object-oriented programming is data and method encapsulation. The users of the class learn how to use the classes through the interface and don't need to know what an object contains exactly, how the data are stored in the object and how the methods are implemented. The second advantage is code reuse. Once a method is implemented in a class, it can be used any time when it is needed by any user. Once a class is defined, it can be used to create specific objects and also can be extended to more specific classes (sub-classes) through the inheritance technique.

1.1 Members of a Class and Access Specifiers

The general syntax for defining a class is following.

```
[access_specifier] class ClassName
{
 //members of the class
}
```

In this book, we mean that the part is optional if it is put into brackets. Each class or a member of a class needs an access specifier which determines whether the class or member can be used or invoked in another class or subclass. There are four different access specifiers: public, private, protected and default (no specifier). If a member of a class is public, this member can be accessed by any code that has access to the class. If a member is private, it can only be accessed by the code in this class. If a member is protected, it can only be accessed by the code in this class and its subclasses. If a member does not have access specifier, it can be accessed by any code in this package.
By access, we mean a call or activation with the dot operator. Their effects are listed in the following table.

Access Specifiers	Default	private	protected	public
Accessible inside the class	Yes	yes	yes	yes
Accessible inside a subclass in the same package	Yes	no	yes	yes
Accessible outside the package	No	no	no	yes
Accessible in a subclass outside the package	No	no	yes	yes

In one file, only one class can be declared public and the file name must be exactly the same as that public class name. By convention, a class name should start with an uppercase letter. A class may have four kinds of members: attributes, constructors, methods and inner classes. An attribute is a data variable that is used to store data of an object. A constructor is a procedure used to create an object of the class. A method is a procedure that defines some behavior of the class. Class names, variable names and method names are called identifiers. Every identifier must follow the following rules.

1). It may contain letters, digits, and underscore.
2). It cannot start with a digit.
3). It cannot be exactly a reserved word.
Traditionally, a dollar sign can also be used in an identifier, but it is not recommended.

1.2 Attributes and Constructors

Java uses attributes to specify the essential data that are needed by every object of a class. For example, each fraction must have a numerator and a denominator. Thus, a Fraction class must have two attributes, one for storing numerator and another for denominator. The general syntax for declaring an attribute is following.

[access specifier] [static] variableType variableName;

To encapsulate data in objects and make sure that data in an object is not modified accidently, an attribute is usually designated as private. When the static keyword is applied to the front of an attribute, the attribute becomes a static or class wide one which has the same value in all objects of the class in one program. That means, if you change the attribute value of one object, the same attribute in the other objects is also changed. An attribute without the static keyword is called an instance one and every object of the class has an independent value for an instance attribute. The variable type and variable name follow the same rules of the local variables. Theoretically, an attribute of a class can be declared anywhere within the braces of a class declaration and outside all the constructors, methods and inner classes. The scope of an attribute starts from the beginning of the class and extends to the entire class, and the scope of a local variable starts from the point where it is declared and extends to the end of the block in which the variable is declared. A local variable and an attribute may have the same name and overlapped scope. To distinguish an attribute from a local variable in their overlapped scope, the attribute needs to be preceded by the keyword this followed by dot and the attribute name. Otherwise, the local variable takes precedence.

A constructor contains a section of code that is used to create an object of the class by initializing the attributes of the class. Here is the general syntax of a constructor.

```
public ClassName(parameterList)
{
 //body
}
```

The access specifier of a constructor is usually public. The constructor name must be exactly the class name. If a class does not have a constructor defined, it receives a default constructor automatically. The default constructor does not have parameters and all the attributes are initialized to the default values of their types. A constructor that has a parameter list is called an explicit one. If a class has an explicit constructor defined, it does not get a free default constructor anymore so a constructor without parameter list needs to be defined if it is needed.
Here is an example for declaring a Fraction class.

```
/**Define a class that has all methods for performing regular operations of
factions
*/
public class Fraction
{
            private double numerator;
            private double denominator;

          /**Default constructor*/
           public Fraction()
           {
            numerator=0.0;
            denominator=1.0;
          }
```

```
        /**Explicit constructor
          *@param numerator the initial value of the numerator of this
          *                        Fraction
          *@param denominator initial value of the denominator of this
          *                          Fraction
          */
        public Fraction(double numerator, double denominator)
        {
          this.numerator=numerator;
          this.denominator=denominator;
        }

}
```

Once a class is defined, an object of the class can be created by using the keyword new and a constructor. For example, the statement

```
   Fraction f1=new Fraction();
```

uses the first constructor to create an object with numerator initialized to 0.0 and denominator initialized to 1.0. The statement

```
  Fraction f2=new Fraction(2.0, 3.0);
```

uses the second constructor to create an object with numerator initialized to 2.0 and denominator initialized to 3.0. As stated earlier, the first statement is still valid if neither of these two constructors is present. Of course, it would initialize both numerator and denominator to 0.0. If the first constructor (default one) were not present in the class definition and the second constructor (the explicit one) were present in this class definition, the default constructor would not be automatic and the first statement would not be a valid one.

1.3 Methods

A method is a procedure with a name that defines some behavior of all objects of its class. The general syntax of a method definition is following.

```
[accessSpecifier] [static] returnType method_name(parameterList)
{
  //body
}
```

A method with the static keyword between the access specifier and the return type in its header is called a class wide or static method. Otherwise, it is called an instance method. An instance method needs data from an object or modifies data from an object so it must be activated/called by an object. A static method may be activated by the class name or an object of the class. The word “activate” means that an object name or class name followed by the dot operator, then a method name with appropriate arguments or data input for the parameters within parentheses. The return type of a method can be either the keyword void or any valid data type. When the return type of a method is a valid data type, it is called a value-returning method. A value returning method must have a return statement in its body that returns a value matching the return type. In Java, this return statement should not be shadowed by another pair of braces in the body of the method. Otherwise, it causes a syntax error. When the return type of a method is the keyword void, it is called a non-value-returning method. A non-value-returning method may not have a return statement although it may use the keyword return to terminate the method. Since a value returning method is to calculate and return a value, a statement that activates a value returning method needs to either use the returned value or store the returned value in a variable. A statement that activates a non-value-returning method cannot do this. A method name is an identifier so it must follow the three rules of identifiers. By convention, a method name starts with a lowercase letter or underscore so it can be distinguished from a class name.

Here are a few methods in the Fraction class.

```
/**Set the numerator to a new value
  *@param numerator the new value of the numerator
  */
public void setNumerator(double numerator)
{
  this.numerator=numerator;
}

/**Convert this Fraction to its equivalent decimal value
  *@return the equivalent decimal value of this Fraction
  */
public double toDecimal()
{
 return (double)numerator/(double)denominator;
}

/**Finds the greatest common divisor of two non-negative integers
```

```
   *@param a the first integer—must be non-negative
   *@param b the second integer—must be non-negative
   */
public static int gcd(int a, int b)
{
  int small=((a>b)?b:a);
   if(small==0) return ((a>b)?a:b);
   for(int i=small; i>1; i--)
      if(a%i==0 && b%i==0) //if both are divisible by i
          return i;
   return 1; //The previous return statement is shadowed by the if statement
 }

/**Simplifies this fraction*/
public void simplify()
{
  int g=gcd(Math.abs(numerator), Math.abs(denominator)); if(g!=1)
  {
   numerator/=g;
   denominator/=g;
  }
}

/**Add this Fraction and another and return the sum
   *@param f the second addend
   *@return the sum of this Fraction and f
   */
public Fraction add(Fraction f)
{
 Fraction f1=new
 Fraction(this.numerator*denominator+this.denominator*numerator,

 this.denominator*denominator);
 f1.simplify();
 return f1;
}
```

The setNumerator and simplify methods are non-value-returning methods. Thus, when the simplify method is used in the second statement of the add method, its activation alone can be used as a complete statement. Neither of these two methods has a return statement in its body. The other methods are value-returning methods. When the gcd method is activated in the first statement of the simplify method, its returned value is stored in a variable g. All these value-returning methods have return statements at the end of their bodies. The gcd method is a static method which does not need data from any object and it can be activated by the class name in the first statement of the simplify method or an object of the class. We need to clarify a point here: we did not use an object, nor the class name to activate the gcd method explicitly in the first statement of the simplify method. Does this create an error? No, If a method is used in another method without the class name or an object activating it explicitly, it is activated by the object or class that activates the outer method implicitly. Because a static method can be activated by either the class name or an object, the class name and dot do not have to precede the gcd method. In that case, the gcd method will be activated by the object activating the simplify method. If an instance method is used in a static method, it must be activated explicitly by an object. Otherwise, when the outer method is activated by its class name, the instance method would be also activated by that class name which causes a syntax error.

Instance methods are also categorized by whether they modify the states of the object activating them or not. An instance method that does not modify the value of any attribute in the object activating it is called an accessor. If an instance method modifies the value of any attribute of the object activating it, it is called a mutator. In our Fraction class, only the simplify and setNumerator methods are mutators because the setNumerator method intends to change the numerator value of the object activating it and the simplify method intends to change the values of both the numerator and denominator. It is important to point out that a constructor is different from a method. A constructor does not have a return type while a method must have a void type if it does not return anything. A constructor must use the exact name of the class while a method can use any name that follows the rules of the identifiers.

In a class, two or more methods are allowed to have the same name. This technique is called method overloading and it allows a programmer to use meaningful names for methods. For example, we may have two multiply methods in the fraction class, one for multiplication of two fractions and another for a fraction being multiplied by an integer. When an overloaded method is activated, the compiler determines which one to use by the signature (name and parameter list) of the methods. It first compares the number of arguments supplied to the method with the number of parameters. If the overloaded methods have the same number of parameters, then the compiler compares the types of the arguments supplied with the corresponding types of the parameters. Thus, two overloaded methods should either have different number of parameters or at least different types of parameters at the same position.

```
   /**Multiply this Fraction by another one
    *@param f the second Fraction used to multiply this Fraction
    *@return the product of this Fraction and f
    */
 public Fraction multiply(Fraction f)
 {
   Fraction f1=new Fraction(this.numberator*numerator,
                                  this.denominator*denominator);
   f1.simplify();
   return f1;
 }
 /**Multiply this Fraction by an integer
    *@param a the integer that multiplies this Fraction
    *@return the product of this Fraction and a
    */
 public Fraction multiply(int a)
 {
   Fraction f1=new Fraction(this.numerator*a, this.denominator);
   f1.simplify();
   return f1;
 }
```

One important method in a class is the toString method which is defined in the Object class. Because every class automatically extends the Object class, the toString method is automatically inherited. The original toString method is an instance one which returns the address of the object activating it. A class may override this method so it formats the object activating it in a string and returns it.

```
 /**Formats this Fraction in the format of numerator/denominator
    *@return the Fraction in the string format: numerator/denominator
    */
 public String toString()
 {
   return this.numerator+”/”+this.denominator;
 }
```

1.4 Primitive vs. Reference Types

Variable types are categorized into two categories, one is called primitive type and the other is called reference type. A primitive type is predefined by the language and is named by a reserved keyword. The Java language supports eight primitive types: byte, short, int, long, float, double, boolean and char. A reference type is defined by a class so its type is the class name. A primitive type variable stores the value assigned to it and a reference type variable stores the address of the object assigned to it.

```
int x=10, y=10;
if(x==y)
   System.out.println(“x and y are equal!”);
else
   System.out.println(“x and y are not equal!”);
```

The message printed out by this section of code is clearly “x and y are equal!”.

```
Fraction f1=new Fraction(2.0, 3.0), f2=new Fraction(2.0, 3.0);
if(f1==f2)
   System.out.println(“f1 and f2 are equal!”);
else
  System.out.println(“f1 and f2 are not equal!”);
```

Since f1 and f2 are holding the addresses of two objects and the comparison operator == compares the values of the operands, the above section of code sends out the message “f1 and f2 are not equal!” although the objects referred to by f1 and f2 contain the same data. If a programmer wants to compare the data stored in the objects referenced by some variables, the equals method from the Object class needs to be overridden.

```
/**Compares if this Fraction is equal to the second one
  *@param f the Fraction that is compared with this Fraction
  *@return true if this Fraction is equal to f, false otherwise
  */
public boolean equals(Fraction f)
{
 return this.numerator*f.denominator==this.denominator*f.numerator;
}
```

After this method is implemented, a programmer may compare the states of two objects referenced by f1 and f2 by using f1.equals(f2). Now, we can compare the state of two Fraction objects by using the equals method:

```
 if(f1.eqauls(f2))
     System.out.println(“f1 and f2 are equal!”);
```

```
else
    System.out.println("f1 and f2 are not equal!");
```

When this section of code is executed, the output is "f1 and f2 are equal!".

1.5 Inheritance and Polymorphism

Inheritance is one of the most important concepts in object-oriented programming. Java language supports inheritance for minimizing duplicates of code and making application code more flexible to change. As the name inheritance suggests, a class is able to inherit characteristics from another class. A class from which another class inherits is called a super class and the class that inherits from a super class is called a subclass.

For example, let's say we make a class called "BankAccount" that represents the general characteristics of a bank account. The state of this class keeps tracking the balance of an account. It has behavior like deposit and withdraw. BankAccount is good for getting overall sense of what makes all bank accounts the same, but it cannot for instance tell about the difference between a checking account and a savings account. The state and behaviors of these two classes will differ from each other in a lot of ways except for the ones in the BankAccount class.

The general syntax for defining a subclass is following.

```
[access_specifier] class subclassName extends superclassName
{
  //body
}
```

Let's define our BankAccount class first.

```
/**Define a simple bank account which has deposit and withdraw method*/
public class BankAccount
{
  private double balance;

  public BankAccount(){ balance=0.0;}
  public BankAccount(double balance){this.balance=balance;}

  public double getBalance(){return balance;}
  public void deposit(double amount){balance+=amount;}
  public void withdraw(double amount){balance-=amount;}
}
```

Now, we are ready to define a SavingsAccount class
that inherits our BankAccount class.

```
/**Define a savings account that has interest and charges some fee for
each transaction*/
public class SavingsAccount extends BankAccount
{
  private double annualRate;
  private int number; //The number of times that the interest is
                              //compounded each year
  private double fee; //Charge for each transaction

  public SavingsAccount(){annualRate=0.0; fee=0.0;}
  public SavingsAccount(double balance, double rate, double fee)
  {
    super(balance);
    annualRate=rate;
    this.fee=fee;
  }

public void deposit(double amount)
{
  super.deposit(amount-fee);
}
  public void compound()
  {
    double interest=getBalance()*annualRate/number;
    super.deposit(interest);
  }
}
```

A subclass inherits all attributes from its superclass although it may not have direct access to the attributes in the superclass. Although an attribute from a superclass can be overridden in a subclass, it is undesirable. Any new attributes defined in a subclass are present only in subclass objects, not in superclass objects.

A subclass constructor needs to initialize the attributes inherited from its super class. Because the super class attributes are private, they cannot be accessed by a subclass constructor. Thus, a superclass constructor needs to be activated in a subclass constructor for this purpose. To activate a superclass constructor in a subclass constructor, the super keyword can be used as in the first statement of the

SavingsAccount explicit constructor. This superclass constructor call has to be the first statement in the subclass constructor. If a subclass constructor does not call/activate one of its superclass constructors explicitly, the default constructor of its superclass is activated automatically first, and then the code of the subclass constructor is executed. If the superclass does not have any constructor, this is not a problem because it automatically gets a default constructor for free. If the superclass has explicit constructors defined without a constructor that does not have empty parameter list, this becomes a syntax error.

A subclass may have three kinds of methods.
1). New methods like the compound method in the SavingsAccount class that don't exist in its super class can be defined. These new methods can only be applied to subclass objects.
2). Methods from the superclass are all inherited by the subclass so they are all present in the subclass. Although the getBalance method is only defined in the BankAccount class, it is also present in the SavingsAccount class. The superclass methods can be applied to both superclass and subclass objects.
3). Methods from the super class can be overridden in the subclass. These methods have exactly the same signature. Whenever the method is activated by an object of the subclass, the overridden method, not the original one, is executed.

Polymorphism is another important concept in object-oriented programming. Polymorphism literally means "many forms". In object-oriented programming concept it refers to the ability of a variable, function or object to take on multiple forms. For example, the deposit methods in BankAccount class and SavingAccont class have exactly the same signature, but different bodies. Since a subclass is a special class of its superclass, a subclass object is allowed to be assigned to a superclass variable. For example,

```
   BankAccount acount1=new SavingsAccount(100.0, 2.0);
```

When a super class type variable refers to a subclass object and is used to activate an overridden method, the method used is determined by the polymorphism principle: the method used is determined by the type of the object, not the type of the variable at runtime a. Thus, the following method call

```
   account1.deposit(500.0);
```

activates the deposit method from the SavingsAccount class, not the BankAccount class. When a superclass variable is used to refer to a subclass object, the variable knows less than the object actually has. So it is a syntax error if you use the variable to activate a subclass method that does not exist in the superclass.

1.6 Exception Handling

An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions. A method may throw an exception to a higher level when the exception cannot or should not be handled at the level of implementation of the method. For example, an exception occurs when a fraction is divided by a zero fraction. In this case, we need to define a ZeroDenominator exception and allow the dividedBy method to throw an exception of that type. A class defines an Exception if it extends an Exception class of the library.

```
public class ZeroDenominatorException extends RuntimeException
{
   public ZeroDenominatorException(String message)
   {
      super(message);
   }

   public ZeroDenominatorException(){};

}

/**Divide this Fraction by another Fraction
  *@param f a Fraction as divisor
  *@return the quotient when this Fraction is divided by f
  */
public Fraction dividedBy(Fraction f) throws ZeroDenominatorException
{
 if(f.numerator==0)
    throw ZeroDenominatorException("Cannot be divided by zero!");
 Fraction f1=new Fraction(this.numerator*f.denominator,
                                            this.denominator*f.numerator);
f1.simplify(); return f1;
}
```

A user who uses this method needs to make decision whether s/he wants to throw the potential exception to another level or handle the exception. If s/he wants to throw the potential exception to another level, s/he can place a throws clause at the end of the header of the method that uses this method. Otherwise, s/he needs to use a try-catch block to handle the exception.

1.7 The Comparable, Comparator Interfaces, Arrays, and Collections Class

The Comparable and Comparator interfaces can be implemented to define orderings for the objects of a class. With these orderings defined by the Comparable or the Comparator interface, a few methods in the Arrays class can be used to sort the elements of the class type in an array, and several methods in the Collections class can be used to sort the elements of the class type in a list. The following are the methods in the Comparable and the Comparator interfaces. Recall that all the methods of the interface need to be implemented if a class implements an interface. The equals method in the Comparator interface does not have to be implemented if it is not needed because every class inherits the Object class and the equals method is defined in this class.

Comparable

Return Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order.

Comparator

Return Type	Method and Description
`int`	`compare(T o1, T o2)` Compares its two arguments for order.
`Boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this comparator.

Again both of these interfaces are generic and the type T represents the type of the elements which need to be ordered. In convention, a positive integer is returned if the first object is greater than the second object, zero is returned if the first object is equal to the second object, and a negative number is returned if the first object is less than the second object. For example, if we want to define an ordering of the objects of our Fraction class by using the Comparable interface, we need to modify the Fraction class by changing its header to public class Fraction implements Comparable<Fraction> and adding the following method to the Fraction class body.

```
public int compareTo(Fraction o)
{
   if(this.toDecimal()>o.toDecimal())
        return 1;
  else if(this.toDecimal()<o.toDecimal())
      return -1;
  return 0;
 }
```

With this part added to the Fraction class, we may use the sort method to sort an array of Fractions, or an ArrayList of Fractions.

```
Fraction[] list1=new Fraction[20];
Random generator=new Random();
for(int i=0; i<20; i++)
   list1[i]=new Fraction(generator.nextInt(100), 1+generator.nextInt(99));
java.util.Arrays.sort(list1);
```

The above sort statement sorts the Fractions in list1 into ascending order according to the ordering defined by the Comparable interface. If a descending order is needed, a class that implements the Comparator interface should be defined and another sort method from the Collections class can be used.

```
public class FractionOrdering implements Comparator<Fraction>
{
  public int compare(Fraction f1, Fraction f2)
   {
      return (-1)*f1.compareTo(f2);
   }
}
```

With this class defined, you may sort an array or ArrayList of Fractions into descending order.

```
ArrayList<Fraction> list2=new ArrayList();
Random gen=new Random();
for(int i=0; i<20; i++)
   list2.add(gen.nextInt(100), 1+gen.nextInt(99));
java.util.Collections.sort(list2, new FractionOrdering());
```

Review Questions

1. What is a class?
2. What is object oriented programming?
3. What are the advantages of object oriented programming?
4. What are the three rules for naming an identifier?
5. What is an attribute of a class?
6. What is the scope of an attribute of a class?
7. What is a local variable?
8. What is the scope of a local variable?
9. Java language allows scope overlap of an attribute and local variable. What happens when there is scope overlap between an attribute and a local variable?
10. What is a constructor?
11. What is a method?
12. What are the differences between a constructor and a method?
13. What are the differences between a value-returning method and a non-value-returning method?
14. What is an accessor?
15. What is a mutator?
16. What are the differences between a static method and an instance method?
17. Complete the following class definition by filling up the blanks.

```
public class Date
{
  private int day;//Data variable for the date
  private int month;//data variable for the month
  private int year;//data variable for the year

  public Date(){day=1; month=1; year=2007;} //Default constructor

  public Date(int day, int month, int year)
  {                                                              }

   public int getDay(){ return day;}
   public int getMonth(){ return month;}
   public int getYear(){ return year;}

  /**Define a method that compares whether this Date is equals another*/
  public
/**Define a toString method that converts this Date object into a string of the
    Format: month/day/year*/
Public



}
```

18. Write the necessary java statement to perform the given tasks.
    1). Create a Date object that contains today's date, declare a Date type variable and assign it to reference the newly created object.
    2). Use a method of the Date class and a standard printing method to get and print out the year of the object created in the previous part of this problem.
19. In the class defined in problem Number 17, which methods are mutators? Which methods are accessors?
20. You may define an ordering of the objects of a class by implementing the interface_____________which has one method _______________________________________ or the interface _______________ which has one method____.
21. What is a reference type variable?
22. What is the difference between a primitive type variable and a reference type variable?
23. What happens when you use the == operator to compare two reference type variables?
24. What is a super class?
25. What is a subclass?
26. A subclass does not have direct access to its superclass private members. How does a subclass initialize its superclass attributes?
27. If a subclass constructor does not activate its superclass constructor explicitly, it activates its superclass default constructor implicitly. What problem may this cause?
28. What is an exception?
29. What makes an object of a class an exception?
30. What are the differences between the Comparable and Comparator interfaces?

Programming Projects

1. A complex number with a real part a and an imaginary part b is written as $a+bi$ where i is called the imaginary unit and $i^2=-1$. For example, $2-3i$ is a complex number with real part 2 and imaginary part -3; 2 is a complex number with real part 2 and imaginary part 0; and $2i$ is a complex number with real part 0 and imaginary part 2.

The sum of two complex numbers is a **new complex number** whose real part is the sum of the real parts of the complex numbers and whose imaginary part is the sum of the imaginary parts of the complex numbers. More precisely, if one complex number has real part a and imaginary part b, and another complex number has real part c and imaginary part d, the sum of these two complex numbers is defined to be $(a + bi) + (c + di) = (a + c) + (b + d)i$. For example,
$(2 + 3i) + (5 - 6i) = (2 + 5) + (3 - 6)i = 7 - 3i$.

The difference of two complex numbers is a **new complex number** whose real part is the difference of the real parts of the complex numbers and whose imaginary part is the difference of the imaginary parts of the complex numbers. More precisely, if one complex number has real part a and imaginary part b, and another complex number has real part c and imaginary part d, the difference of these two complex numbers is defined to be $(a + bi) - (c + di) = (a - c) + (b - d)i$. For example,
$(2 + 3i) - (5 - 6i) = (2 - 5) + (3 - (-6))i = -3 + 9i$.

The magnitude of a complex number is a **real number** that can be found by taking the square root of the total of the square of the real part and the square of the imaginary part. More precisely, the magnitude of the complex number $a + bi$ is defined to be $\sqrt{a^2 + b^2}$. For example, the magnitude of $4 - 3i$ is equal to
$\sqrt{4^2 + (-3)^2} = \sqrt{25} = 5$.

Two complex numbers are equal if and only if their real parts are equal and their imaginary parts are also equal.

The product of two complex numbers is a **complex number** that is defined by
$(a+bi)(c+di) = (ac-bd) + (ad+bc)i$. For example, $(2+3i)(3-2i) = 12+5i$.

The quotient of two complex numbers is a new **complex number** that is defined by
$\frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{c^2 + d^2} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$. For example,

$$\frac{1 + 2i}{3 - 4i} = \frac{(1 + 2i)(3 + 4i)}{25} = -\frac{1}{5} + \frac{2}{5}i.$$

Define a ComplexNumber class.

1) Your class should have one default constructor (without parameter list)and one explicit constructor. In the default constructor, each attribute is assigned to 0.0.
2) Your class should have methods: ***add, subtract, magnitude, equals, times, divides and toString***.
3) The ***add*** method can be used to add two complex numbers. The ***subtract*** method can be used to subtract a complex number from another. The ***magnitude*** method can be used to find the magnitude of a complex number.

The ***times*** method can be used to find the product of two complex numbers. The ***divides*** method can be used to find the quotient of two complex numbers. The ***equals*** method can be used to compare whether two complex numbers are equal or not.

4) Define the ***toString*** method so that a complex number can be printed out in the format of $a + bi$ when it is supplied to a print out method (be careful about a negative imaginary part).
5) Hint: Your class should have two double type attributes. The return type of the ***add*** method should be ***ComplexNumber*** if your class name is ***ComplexNumber***. The return type of the ***magnitude*** method should be ***double***. The return type of your ***equals*** method should be ***boolean***.
6) Don't forget to include proper comments, appropriate indentation and alignment.
7) Write another class with main method to test all constructors and methods of your class.
8) Apply javadoc to your ComplexNumber class and produce an html document of the class.
9) Hand in your homework in a folder with pockets. Write your name carefully in the front cover of your folder. The folder should contain a hard copy of your classes, a hard copy of the document generated by the javadoc, a hard copy of sample output of your test class and a disk containing your programs.

2. An n-by-m matrix is an n-by-m array of numbers. For example, $\begin{pmatrix} 2 & 5 \\ -3 & | \; 8 \\ 7 & 5 \end{pmatrix}$ is a 3-by-2 matrix. The position of an entry of a matrix is given by its row and column. For example, the entry of this matrix at position (1, 1) is 2 and the entry at position (2, 1) is -3. If two matrices have the same dimension, they

   can be added or subtracted by adding or subtracting the corresponding entries. For example,

   $$\begin{pmatrix} 2 & 5 \\ -3 & 8 \\ 7 & 5 \end{pmatrix} + \begin{pmatrix} -3 & 10 \\ 2 & 8 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} -1 & 15 \\ -1 & 16 \\ 7 & 5 \end{pmatrix}$$

   and $\begin{pmatrix} 2 & 5 \\ -3 & 8 \\ 7 & 5 \end{pmatrix} - \begin{pmatrix} -3 & 10 \\ 2 & 8 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 5 & -5 \\ -5 & 0 \\ 7 & 5 \end{pmatrix}$.

   A 1-by-n matrix (a row with n entries actually) can be multiplied by an n-by-1 matrix (a column with n entries actually) to get a real number. This number is obtained by summing up the first number in the row times the first number in the column, the second number in the row times the second number in the column, …, and the last number in the row times the last number in the column. For example,

$$\begin{pmatrix} 5 & 8 & 3 \end{pmatrix} * \begin{pmatrix} -1 \\ 2 \\ -3 \end{pmatrix} = 5 * (-1) + 8 * 2 + 3 * (-3) = 2.$$

An n-by-m matrix times an m-by-p matrix is an n-by-p matrix (the number of columns in the first matrix must be equal to the number of rows in the second matrix) whose entry at the (i, j) position for 1<=i<=n and 1<=j<=p is the result of the ith row of the first matrix times the jth column of the second matrix. For example,

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} * \begin{pmatrix} 2 & 5 \\ -3 & 8 \\ 7 & 5 \end{pmatrix} = \begin{pmatrix} 1 * 2 + 2 * (-3) + 3 * 7 & 1 * 5 + 2 * 8 + 3 * 5 \\ 3 * 2 + 2 * (-3) + 1 * 7 & 3 * 5 + 2 * 8 + 1 * 5 \end{pmatrix}$$

$$= \begin{pmatrix} 17 & 36 \\ 7 & 36 \end{pmatrix}.$$

In this example, the first matrix has 3 columns so the second one must have 3 rows that makes it possible for each row to multiply each column. Since the first matrix has 2 rows and the second matrix has 2 columns, the resultant matrix has 2 rows and 2 columns. Two matrices are equal if they have the same dimensions and their corresponding entries are equal.

1) Design a **Matrix class** whose attributes include the number of rows, the number of columns and a two dimensional array that is used to refer to the matrix. The constructor of this class creates the required array and initializes every entry to 0.
2) Write a **setEntry** method that takes two int parameters as the position of an entry and a double type parameter, and sets the given number to the position.
3) Write an **add** method that adds this matrix (the matrix that calls/activates the method) and another matrix. If the dimensions of two matrices don't match, your method should throw a dimensionMismatch exception.
4) Write a **subtract** method that subtracts a matrix from this matrix. If the dimensions of two matrices don't match, your method should throw a dimensionMismatch exception.
5) Write a **multiply** method that finds and returns the product of this matrix and another matrix if they have appropriate dimensions. If the dimensions of the matrices are not appropriate for multiplication, your method should throw a dimensionMismatch exception.
6) Write an **equals** method that checks whether this matrix is equal to another matrix.
7) The transpose of an n by m matrix is an m by n matrix whose value at the (i, j) position is the value at (j, i) position of the original matrix. Define and implement a transpose method that finds the transpose of this matrix.

8) The Arrays class has a static **sort** method for sorting arrays of objects if an ordering of the objects is defined. Let's make up an ordering for matrices: one matrix is less than another matrix if the sum of the absolute values of the entries of the first matrix is less than the sum of the absolute values of the entries of the second matrix. Your matrix class should implement the **Comparable** interface so that natural ordering is defined by our agreement above.
9) You need to also implement the **Comparator** interface so that the opposite ordering of matrices is defined.
10) Write a test class that tests the constructor and every method you have defined in the matrix class.
11) Declare an array of matrices and fill it up with matrices. Use a sort method in the Arrays class to sort your array in ascending order.
12) Use a sort method from the Arrays class to sort your matrix array in descending order.
13) Design and implement a DimensionMismatch exception class and use it when two matrices don't have appropriate dimensions in the methods such as add, subtract and multiply.
14) Make sure that you use professional documentation (comments) so that javadoc can be applied.

3. Poker Simulator. Implement a simulation of a popular casino game called video poker. The card deck contains 52 cards, 13 of each suit. At the beginning of the game, the deck is shuffled. The top five cards of the deck are presented to the player. The player can reject none, some, or all of the cards. The rejected cards are replaced from the top of the deck. Now the hand is scored. Your program should pronounce it to be one of the following:

- No pair—The lowest hand, containing five separate cards that do not match up to create any of the hands below.
- One pair—Two cards of the same value, for example, two queens.
- Two pairs—Two pairs, for example, two queens and two 5's.
- Three of a kind—Three cards of the same value, for example, three queens
- Straight—Five cards with consecutive values, not necessarily of the same suit, such as 4, 5, 6, 7, and 8. The ace can either precede a 2 or follow a king.
- Flush—Five cards, not necessarily in order, of the same suit.
- Full house—Three of a kind and a pair, for example, three queens and two 5's.
- Four of a kind--Four cards of the same value such as four queens.
- Straight flush—A straight and a flush: Five cards with consecutive values of the same suit.
- Royal flush--The best possible hand in poker: a 10, jack, queen, king, and ace, all of the same suit.

Now, you are required to use a List (ArrayList or Vector) to hold your deck of the cards and an array to hold the top five cards drawn from the deck. Because each of your cards belongs to one of the four suits and has a value, you need a Card class that has two attributes. Your Card class is required to implement the Comparable interface so that a static shuffle method in the Collections class can be used to shuffle your deck and a static sort method in the Arrays class can be used to sort your hand of five cards. It also needs at least the following methods.

getSuit—returns the suit of the card.

getValue—returns the value of the card.

How do you evaluate a hand? Order the hand by using the static sort method from the Arrays class first and then make the comparison from the highest hand (Royal flush) to the lowest hand (No pair)..

Do not forget to have appropriate indentation and professional documentation so that you can apply javadoc to your class to produce a document of your class.

Shuffle your cards, pick the first five to display, ask the player which cards of the five should be discarded, discard the cards, pick the same number of the cards from the top of the deck, evaluate and report the rank of the hand.

Make sure that you understand the classes needed for this project: a Card class, a Poker class that has a deck attribute, a hand attribute and various constructors and methods, and a class with the main method.

4. The existing types for integer in Java language don't reserve enough memory for scientific computation. Thus, we need to define a new type which may store an integer of hundreds of digits. Design a BigInteger class that uses an array of sigle digits as storage and provides all the operations such as addition, subtraction, multiplication, division and modulation, and all operators such as >, <, >=, <= and ==.

   Make sure that you understand the number of digits of an integer may not be the length of the array and you need to enlarge the array size when needed. Because your class will be used to handle integers that cannot be handled by the existing types, your constructor should take a numerical string as input. It may be convenient to store an integer in reversed order for implementing the operations.

Chapter Two
Container Classes

A container class is one whose objects store large groups of elements. In the Java library, there are two families of container classes: Collection and Map. Collections are also divided into two families: List and Set.

2.1 Lists and Iterator

A List has order and may have duplicates of any element. Elements can be inserted or accessed by their position in the list, using a zero-based index. In addition to the methods defined by **Collection**, List<E> defines some of its own, which are summarized in the following Table.

SN	Methods with Description
1	**void add(int index, E e)** Inserts e into this list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	**boolean addAll(int index, Collection c)** Inserts all elements of c into this list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	**E get(int index)** Returns the element stored at the specified index within this list.
4	**int indexOf(E e)** Returns the index of the first instance of e in this list. If e is not an element of the list, -1 is returned.
5	**int lastIndexOf(E e)** Returns the index of the last instance of e in this list. If e is not an element of the list, -1 is returned.
6	**ListIterator listIterator( )** Returns an iterator with an imaginary cursor at the beginning of this list.
7	**ListIterator listIterator(int index)** Returns an iterator to the invoking list that begins at the specified index.

8	**E remove(int index)** Removes the element at position index from this list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one
9	**E set(int index, E e)** Assigns e to the location specified by index within this list.
10	**List subList(int start, int end)** Returns a list that includes elements from start to end-1 in this list. Elements in the returned list are also referenced by the invoking object.

An iterator is an object that enables a programmer to traverse a container, particularly lists. An iterator performs traversal and also gives access to the data elements in a container. It behaves like a database cursor. In java language, an iterator also performs removal of elements from a container. The following are the methods in the iterator interface.

Methods

Modifier and Type	Method and Description
`boolean`	**`hasNext`**`()` Returns `true` if the iteration has more elements.
`E`	**`next`**`()` Returns the next element in the iteration and advances the cursor one element up.
`void`	**`remove`**`()` Removes from the underlying collection the last element returned by this iterator (optional operation).

These tables involve the concept of generic classes which should have been covered briefly in a previous course. Generic programming will be covered carefully in next chapter. In this table, E is the type of the elements in the underlying collection. In the library, four classes, ArrayList<E>, LinkedList<E>, Vector<E> and Stack<E> implement the List interface. ArrayList<E> should have been covered before this course.

Here is a section of code that shows how some of the List methods can be used:

```
LinkedList<String> list=new LinkedList();
        list.add(0, "This");
        list.add(1,"is");
        list.add(2, "a");
        list.add(3, "list");
        for(ListIterator i=list.listIterator(); i.hasNext(); )
           System.out.print(i.next()+(i.hasNext()?", ":"."));
        System.out.println();
```

A list can be defined by using an array or a sequence of linked nodes. The ArrayList class in the library is defined by using an array and you will be asked to define your own ArrayList as homework.To understand how a LinkedList works. Let’s define our own LinkedList class. A linked list is a container which contains a sequence of nodes and each node stores an element and a reference to the next node in the sequence. A linked list should be a generic class that has an underlying type and may be used by a programmer to store any type of data. Because the concepts of generic class will be covered in next chapter, a linked list of Objects is going to be defined in this chapter and will be modified into a generic class in next chapter.

To define our linked list, our first task is to define a Node class whose object may contain an Object and a link to next Node.

```
public class Node
{
  private Object data;
  private Node link;

   public Node(){link=null;}
   public Node(Object data, Node link){this.data=data; this.link=link;}
   public Object getData(){return data;}
   public Node getLink(){return link;}
   public void setData(Object data){this.data=data;}
   public void setLink(Node link){this.link=link;}
}
```

Now, we are ready to define our own linked list class. We have choices to define a list using linked list technique or define an ordered linked list. In this section, we simply define a list and leave the ordered linked list to students as homework. The essential attribute of this list is the head that points to the first node in the list. The second attribute that points to the last node in the list and the third attribute that represents the number of elements in the list are not necessary, but they are handy and make the code more efficient.

```
/**Define a list with regular operations using the linked list technique*/
public class MyLinkedList
```

```
 {
  private Node head; //Refers to the first Node
  private Node last; //Refers to the last Node
  private int count; //Total number of elements

  public MyLinkedList(){head=null; last=null; count=0;}

/*Rreturns the length of the list
   *@return the length of the list
   */
public int size(){return count;}

/**Appends an element to the end of the list
      *@param e the element added to the list
      */
  public void add(Object e)
 {
  Node newNode=new Node(e, null);
  if(head==null)//if it is empty
  {
     head=newNode;
   last=head;
   }
   else
   {
     last.setLink(newNode);
    last=newNode;
   }
   count++;
 }
/**Insert the element at the beginning of this list
   *@param e the element inserted to the list
  */
public void addFirst(Object e)
{
  head=new Node(e, head);
  if(last==null) last=head;
  count++;
}

/**Insert the element to the specified position
```

```
   *@param index the position at which the element is inserted. If index is
   *                      larger than the size, the element is appended to the end
   *                      of the list
   *@param e the element to be inserted
   */
 public void add(int index, Object e)
 {
   if(index>count) add(e);
   else if(index==0) addFirst(e);
   else
   {
    Node previous=head, current=head;
    for(int i=1; i<index; i++)
    {
      previous=current;
      current=current.getLink();
     }
      Node newNode=new (e, current);
      previous.setLink(newNode);
      count++;
    }

    /**Remove the first element in the list*/
    public void removeFirst()
    {
      if(head!=null){ head=head.getLink(); count--;}
      if(count==0) last=null;
    }

    /**Check if the list contains this element or not
       *@param e the element being checked
       *@return true if e is in the list, false otherwise
       */
    public boolean contains(Object e)
    {
      for(Node n=head; n!=null; n=n.getLink())
         if(n.getData().equals(e)) return true;
      return false;
    }
    public String toString()
    {
      String list= “[“;
      for(Node n=head; n!=null; n=n.getLink())
        list+=n.getData()+((n.getLink()==null)? “]”: “, ”);
    }
 }
```

It is important to point out that using the Object type as the data type of a container class is not a good practice. One direct consequence of this practice is that any one object of MyLinkedList class may contain many different types of objects. For example, the following statements are valid. This problem will be resolved after we study how to define generic classes in Chatper Three.

```
MyLinkedList list=new MyLinkedList();
list.add(new Fraction(2, 3));
list.add(“A String”);
list.add(new Double(2.5));
```

Comparing how an ArrayList and a LinkeList are implemented, we can find the advantages and disadvanteds of ArrayList. and LinkedList. An array uses direct access to an element if we know its index so the get(int i) method of the ArrayList class is very efficient. A LinkedList is remembered by a reference to its header node so we needs to start from its header node to get to its i-th node if we want to access its i-th node so its get(int i) method is inefficient comparing with the one in ArrayList. An array has a fixed length so we need to allocate a longer array and transfer all the elements from the old array to the new array when the array used in the ArrayList is full. While a LinkedList can never be full as long as the machine allows so a LinkedList is more efficient than an ArrayList in this aspect. When we need to add an element to or delete an element from an array, we need to shift all the elements to the right or left, respectively. While adding an element to or deleting ane element from a LinkedList only involves changing the values of a couple of links. Therefore, LinkedLists are also more efficient than ArrayLists in this aspect.

2.2 Sets and Maps

A set is different from a list since it follows the mathematical theory that a set does not have order and does not allow duplicates of any elements. The most important difference is the add method that first checks whether the element to be added is already in the set or not, adds the element to the set and returns true if the element is not already in the set, and leaves the set unchanged and returns false otherwise. The following are the major methods of the Set<E> interface in the library.

boolean add(E e)—adds e to this set if it is not already in the set and returns true,

and returns false otherwise.
boolean addAll(Collection<? extends E> c)—Adds all of the elements in c to this set if they are not already in the set.
void clear()—Removes all the elements from this set.
boolean contains(E e)—returns true if this set contains e and false otherwise. boolean containsAll(Collection<?> c)—Returns true if this set contains all of the elements in c and false otherwise.
boolean isEmpty()—Returns true if this set contains no elements and false otherwise.
boolean remove(E o)—Removes o from this set if it is in the set and returns true, and returns false otherwise.
boolean removeAll(Collection<?> c)—Removes all of the elements contained in c from this set.
int size()—Retruns the number of elements in this set.
Object[] toArray()—Returns an array containing all of the elements in this set.

The addAll methods has a Collection as its parameter which requires the type of elements in the collection to be E type or a type that extends E type.

In the Java library, The HashSet<E> and TreeSet<E> classes implement the Set<E> interface.
Example: Write a simple program that reads from a file, counts the number of distinct words and prints all the distinct words to the screen.

```
public class WordSet
{
  public static void main(String[] args)
  {
   try{
          File f=new File(args[0]);
          BufferedReader in=new BufferedReader(new FileReader(f));
          Set<String> set=new HashSet();
          String line=null;
          StringTokenizer words=null;
          while((line=in.readLine())!=null)
          {
            words =new StringTokenizer(line, " ,.?!-\n\r\t");
            while(words.hasMoreTokens())
                    set.add(words.nextToken().toLowerCase());
          }
          System.out.println(args[0]+ " has " +set.size()+ "distinct words+
                                        ": "+set);
        }catch(Exception e){e.printStackTrace();}
    }
  }
```

A Map in Java is a data structure that keeps association between a set of keys and a collection of values. Every key in a map has a unique value, but a value may be associated with several keys. The Map interface in the library is a generic one with two type parameters which is implemented by the HashMap<K, E> and TreeMap<K, E> classes. The hashing technique will be covered in next section and the tree storage should be covered in an algorithm class. Briefly speaking,

when a pair is being added to a HashMap, it finds a code of the key and uses the code to find a position for the pair so this is an efficient storage. The following are the major methods in the Map<K, E> interface.

void clear()—Removes all the pairs from this map.
boolean containsKey(K key)—Returns true if this map contains a pair for key and false otherwise.
boolean containsValue(E value)—Returns true if this map contains a pair for value, and false otherwise.
V get(K key)—Returns the value to which key is associated or null if key is not in this map.
boolean isEmpty()—Returns true if this map contains no key-value pairs, and false otherwise.
Set<K> keyset()—Returns a Set view of the keys contained in this map.
E put(K key, E value)—Associate this pair and place it to this map.
E remove(K key)—Removes the pair with key from this map and returns the associated value.
int size()—Returns the number of key-value pairs in this map.
Collection<V> values()-Returns a Collection view of the values in this map.

Here is a section of code that uses HashMap<K, E>.

```
Map myMap=new HashMap<String, char>();
myMap.put("Mike", 'A');
myMap.put("Robert", 'B');
myMap.put("David", 'C');
System.out.println("Mike's grade is "+myMap.get("Mike"));
System.out.println("The grade list: "+myMap);
System.out.println("Here is the list of all students:
"+myMap.keySet()); System.out.println("Here is the list of
all grades "+" assigned: "+myMap.values());
```

2.3 Hashing Technique

Hashing is the transformation of an object into an integer value that represents the original object. The integer value is called a hashcode of the object. The purpose of hashing is to use the hashcode for the memory location of the object. A **hash table** (also hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A **hash table** uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

he issues of implementing a hash table include hashcode collision and hashcode out of array bound. Hashcode collision means that two or more objects have the same hashcode. The second issue is that a hashcode is out of the bound of the array used. A simple way to solve the first problem is to store a list in each position of the array used. The second problem can be solved by using the array length to modulo the hashcode produced. In the ideal scenario of this technique, every position of the array has one and only one element. In the worst case when we have a bad hashing function, all the elements are scooched into one list. The following is an example for hashing technique.

```
public class MyHashSet
{
    private MyLinkedList[] set;
    private int capacity;

    public MyHashSet(int cap)
    {
     capacity=cap;
     set=new MyLinkedList[capacity];
     for(int i=0; i<capacity; i++)
         set[i]=new MyLinkedList();
    }
```

```
  /**Hash function
   */
  public int hashCode(Object item)
  {
                    int h=item.hashCode();//call hashCode method from
                                                 //Object class
    if(h<0) h=-h;
    if(h>=capacity) h%=capacity; return h;
  }
  public void add(Object item)
  {
   set[hashCode(item)].addFirstt(item);
  }

  public boolean contains(Object item)
  {
   return set[hashCode(item)].contains(item);
  }

  public String toString()
   {
      String s="[";
      for(int i=0; i<capacity; i++)
          if(!set[i].isEmpty())
              s+=set[i]+", ";
      s+="]";
      return s;
   }
}
```

Review Questions

1. What is a container class?
2. What are the two families of container classes?
3. What are the differences between a list and a set?
4. What is an iterator?
5. What is the return type of the listIterator() method from the list class?
6. What does the next() method in the iterator interface do when it is activated?
7. What is a linked list?
8. What is the advantage of using a linked list over an array?
9. What is a map?
10. What is the difference between the add(E e) method in a list and a set?
11. What is hashing?
12. What is a hash table?
13. What is a hashcode?

14. What are the two typical issues for creating a hash table and how are they resolved?
15. What is the advantage of using a hash table for astorage?
16. What is the worst case and best case for hashing?

Programming Projects

1. We have been using the ArrayList class int the library. Its name means that the list is defined by using an array. Use array to define your own ArrayList class named as MyArrayList that may store a list of objects and has the following methods.
   Object get(int index)—Returns the number of objects in this list.
   void add(Object object)—Appends the given object to the end of this list and increases the size of the list by one.
   void add(int index, Object object)—Inserts the given object at the specified position by index in this list and increase the size of the list by one.
   void set(int index, Object object)—Replaces the object at the given index by the given object in this list.
   void clear()—Emptied this list.
   boolean isEmpty()—return true if this list is empty and false otherwise.
   void remove(Object object)—Removes the first copy of the given object from this list.
   Object remove(int index)—Removes and returns the object at the given index of this list.

   Your array attribute has a default length when an object of your class is created. You need to check the length of the array and increase its length if it is full when you define your add method. Because this is not an ordered list, you may shift the objects over one position or place the last object of the list to the given position when you define the remove methods.
2. A stack in Java is a data structure that contains a stack of objects and follows the last-in-first-out policy. In the Java library, a generic class Stack<E> is defined and has the following methods:
   boolean empty()—Tests if this stack is empty
   E peek()—Returns the object at the top of this stack without removing it from the stack
   E pop()—returns and removes the object at the top of this stack
   E push(E e)—pushes the element e onto the top of this stack
   int search(Object e)—Returns an integer as the 1-based distance of the top element to this element

Use a sequence of linked nodes to define our own stack class with the above methods. Because generic class has not been covered, you may use Object type as the data type in the stack.

3. Write a program that produces a sales report by reading information about a company's sales from a text file. The program then writes to the screen the k largest sales made by each salesperson, for an arbitrary input k.

Part I. Implement a class OrderedList that uses a linked sequence of nodes to maintain an ordered list, i.e., the nodes are ordered so that the data values they contain are in sorted order (from smallest to largest) as you traverse the list from its first node to its last. You need to assume that the type used to instantiate the node and OrderedList is a type that impelements the Comparable interface

The **public interface** of the OrderedList class must include:

a) a method **remove** that removes the first instance of a given item from the list
b) a method **kLargest** that returns a new OrderedList consisting of copies of the k largest elements of the list. If k is larger than the length of the list, return the list.

c) a method **get** that returns a copy of the k-th item in the list for given integer k. If k is larger than the length of the list minus 1, throw an out of bound exception.

d) a method **insert** that creates a new node with a given item and inserts it into its **correct** position in the list.

e) an **addition method** implemented so that (list1.add(list2)) returns a new ordered list containing copies of the items in both list1 and list2. Neither list1 nor list2 is modified by this operation. The construction of the new list must be done **as efficiently as possible**, i.e., by traversing through the nodes of list1 and list2 exactly once and without repeatedly having to search for the location where items should be added to the new ordered list. Thus you cannot simply cycle through both lists and repeatedly call the new list's insert function on the items so encountered. (Hint: Think about how you might take 2 piles of test papers that are each in sorted order and efficiently use them to create a new pile containing all the papers in sorted order, without repeatedly scanning through either of the original piles from the beginning.)

f) Override the toString method so that a list is formed into a string in the format:
[item1, item2, …]

You are free to implement private functions for use by the public ones.

Your solution should adhere to basic principles of object-oriented programming. The code should be neatly organized, easy to read and understand, with correct indentation, reasonable choices for identifiers, and internal documentation to explain non-obvious details of the logic or its implementation.

Part II. You will use the ordered list class to solve the sales report problem. To keep track of salespeople and their sales figures you'll need a small class called
Salesperson that stores a salesperson's employee id number (an int), first name, last name, and an **ordered list** of all her sales amounts (float values) for the period of time covered in the input file. This class must implement the Comparable interface so that a collection of these objects can be sorted by the id values in these objects. It also needs to define the **equals** method that will be used to find a Salesperson object with a certain id in a collection.
The input file, sales.txt, consists of annual summaries of each salesperson's sales figures. The number of these summaries is not specified in the file and will not be supplied by the user of the program. Write an input loop that will read and process all the information in the file. Each summary will have the following format: the first line will contain the year, sales person's id, first name, last name, and the number of sales made by that salesperson in that year all on one line, separated by one or more spaces. That will be followed by the dollar amounts of those sales on a second line, separated by one or more spaces. Each
summary after the first one has this same format but is separated from the previous one by a blank line.
Create an ordered list of sales amounts from each annual summary in the input file. If the current summary is the first one for a particular salesperson, create a Salesperson object that contains the salesperson's id, first and last name and the **ordered list (the ordered list you have defined)** of sales amounts (the year information in the file is not used). Add this object to an initially empty **list** (**use the LinkedList from the Java library**) of these objects. If the current summary is not the first one for a particular salesperson, i.e, that salesperson is already in the list, modify the object in the list by merging the current ordered list of sales amounts with the one already associated with that salesperson in the list (use the ordered list class' addition method). After processing the input file, there should be one salesperson object in the list for each different salesperson regardless of how many annual summaries that salesperson had in the input file.
After processing the entire input file and creating the list of salesperson objects, write the sales report to the screen. First, read the number k from the user. This integer specifies how many sales amounts should be included in the report for each salesperson. Sort the list (by employee id) and use an **iterator** to traverse the list and process each salesperson. The report should contain one line for each salesperson. Each line should start with the id, then the first and last name, and then the k largest sales amounts for that salesperson in the time covered by the input file. List the k sales amounts in order from **largest to smallest** (assume all salespersons have at least k sales).

4. Use the linked list technique to design a BigInteger class specified in problem number 4 of the previous chapter. For efficiency, each node in the list is required to store three digits. So you need to be careful about the leading zeros in a node when you concatenate them back to your integer.

5. Design a Dictionary class by using the hashing technique. You are required to use an array of linked list with a length of 50,000 (assuming that we have at most 50,000 words). You need to use the hashCode method from the Object class to compute the hash code of each word (modulo it if necessary) and store the word and its description in the linked list at the index (of the array) given by the code. Your Dictionary class has at least three public methods: static int hashCode(String word), void add(String word, String description) and lookFor(String word). The hashCode method of this class uses the hashCode() method from the Object class to find a hashCode of the given word and modulo it if necessary. The lookFor(String word) method should return the word and **all its descriptions** (It may have multiple meanings) in an appropriate format (numbered or bulleted items).

   1) You are required to have a small class whose objects hold a word and its description. Your linked list should be a list of objects of this class. Each word is the key for hashing.

   2) You are required to include standard comment for each class and method. 3). Align the braces up vertically and have appropriate indentation.

Chapter Three
Generic Programming

Generic programming is the creation of programming constructs that uses data type parameters to allow the users to use them with different types. When a programmer defines a container class, the programmer may not know what type of data the users may want to put into this container. One user may want to have a linked list of integers, and another may need to have a linked list of strings. When a programmer implements a method for some other users to use, the programmer may not know what types of data the future users need to handle using this method. Generic programming is used to solve this problem.

3.1 Generic Methods

A generic method is one that has a list of type parameters within angle brackets between the method modifiers and the return type. The type parameter allows the method to handle multiple types of data when used. For example, you may want to define a method that finds the larger one of two quantities. The two quantities could be two integers, two strings, two fractions and etc. You certainly don't want to define such a method for each type of these data. With generic programming technique, you may use a type parameter to represent the data type and the compiler infers the type when the method is activated.

```
/**Compares two elements of the same type and returns the larger one*/
public static <T> T larger(T a, T b)
{
  return ((a.compareTo(b)>0)?a:b);
}
```

When this method is used, no more extra work than a regular method is needed. The compiler infers the type of the data from the type of the argument supplied to the method parameters. The method uses the compareTo method from the Comparable interface. If the argument supplied to the parameters is not a type implementing the Comparable interface, an error would occur. To solve this problem, you may place a bound or bounds on the type parameter. To declare a bound for a type parameter, list the type parameter's name, followed by the extends keyword, followed by its *upper bound*. It must be noted that it must be the extends keyword, not implements keyword even it is an interface.

Now, we can modify the above method to the following.

```
/**Compares two elements of the same type and returns the larger one*/
```

```
public static <T extends Comparable<T>> T larger(T a, T b)
{
  return ((a.compareTo(b)>0)?a:b);

}
```

If multiple type parameters are needed, just place them in the angle brackets between the modifier and the return type and separate them by using commas. If multiple upper bounds are needed for one type parameter, just place them following the extends keyword within the angle brackets and connect them using the ampersand symbol.

Since the compiler infers the type for the type parameter from the type of the argument supplied to the method parameter, the type parameter of a generic method must appear in the method parameter list theoretically.

3.2 Generic Classes

Similar to a generic method, a generic class is one that uses one or more type parameters to allow the users to use the class with flexible types of data. For example, we defined a linked list class in section 2.1. To allow different objects of the linked list to contain different types of data, we used Object for our data type. The problem is that one object of the linked list may allow different types of data. The syntax for defining a generic class is very simple: place a type parameter list within angle brackets following the class name. If multiple type parameters are needed, they should be separated by commas. Once a list of type parameters is entered in the brackets following the class name, these type parameters can be used through the class definition as they are regular types. For example, we can modify our Node class in section 2.1 to the following.

```
public class Node<E>
{
  private E data;
  private Node link;


   public Node(){link=null;}
   public Node(E data, Node link){this.data=data; this.link=link;}
   public E getData(){return data;}
   public Node getLink(){return link;}
   public void setData(E data){this.data=data;}
   public void setLink(Node link){this.link=link;}
}
```

When a generic class is instantiated out of the class definition, a reference type (a type defined by a class) needs to be supplied to each type parameter. For creating an object of a generic class, you may simply place a pair of empty angle brackets between the class name and the parentheses containing the arguments. For example, we may have the following statement using the above generic Node class.

Node<String> node=new Node<>(“Advanced Java”, null);

Now, we can modify our MyLinkedList class to a generic class.

```
public class MyLinkedList<E>
{
  private Node<E> head; //Reference to the first Node
  private Node<E> last; //Reference to the last Node
  private int count; //Total number of elements

  public MyLinkedList(){head=null; last=null; count=0;}

/**returns the length of the list
   *@return the length of the list
   */
public int size(){return count;}

/**append an element to the end of the list
   *@param e the element added to the list
   */
  public void add(E e)
 {
  Node<E> newNode=new Node<>(e, null);
  if(head==null)
  {
     head=newNode;
   last=head;
   }
   else
   {
     last.setLink(newNode);
     last=newNode;
   }
   count++;
 }
/**Insert the element at the beginning of this list
   *@param e the element inserted to the list
  */
public void addFirst(E e)
{
  head=new Node<>(e, head);
  if(last==null) last=head;
  count++;
}

/**Insert the element to the specified position
   *@param index the position at which the element is inserted. If index is
   *                      larger than the size, the element is appended to the end
   *                      of the list
   *@param e the element to be inserted
   */
public void add(int index, E e)
{
  if(index>count) add(e);
  else if(index==0) addFirst(e);
  else
  {
   Node<E> previous=head, current=head;
   for(int i=1; i<index; i++)
   {
     previous=current;
     current=current.getLink();
    }
     Node<E> newNode=new Node<>(e, current);
     previous.setLink(newNode);
     count++;
   }

   /**Remove the first element in the list*/
   public void removeFirst()
   {
     if(head!=null){ head=head.getLink(); count--;}
     if(count==0) last=null;
   }
   /**Check if the list contains this element or not
     *@param e the element being checked
     *@return true if e is in the list, false otherwise
     */
   public boolean contains(E e)
   {
     for(Node<E> n=head; n!=null; n=n.getLink())
        if(n.getData().equals(e)) return true;
     return false;
   }

   public String toString()
   {
     String list= "[";
     for(Node<E> n=head; n!=null; n=n.getLink())
       list+=n.getData()+((n.getLink()==null)? "]": ", ");
   }
}
```

Now, we can use different objects of this class to store different type of data.

```
MyLinkedList<Double> numbers=new MyLinkedList<>();
MyLinkedList<String> students=new MyLinkedList<>();
numbers.addFirst(2.0);
numbers.addFirst(3.0);
numbers.addFirst(4.0);
students.addFirst("Mike");
students.addFirst("Bob");
students.addFirst("David");
```

Review Questions

1. What is generic programming?
2. Why do we need to use generic programming?
3. What makes a method a generic method?
4. How to place a bound to a type parameter?
5. How is a type parameter of a generic method initialized when the method is activated?
6. What is the syntax for defining a generic class?
7. How to instantiate an object of a generic class?

Programming Projects

1. Project number 1 in chapter two was not required to use generic programming. Modify the stack class there to a generic class.
2. A Queue is a container class that follows the "first in, first out" policy: only the first element (the one in the front) in the Queue can be seen and removed by the user and data can only be inserted to the rear (after the last one) of the queue. Use the Node class defined in this chapter and the linked list technique to define a **generic** Queue class that has the following methods:

   1). void queue(T obj)—places obj to the rear of the Queue

   2). T deQueue()—retrieves and removes the data in the front of the Queue

   3). int size()—returns the number of pieces of data in the Queue

   4). T peek()—returns the data at the front of the Queue without removing it or returns null if the queue is empty.

   5). boolean contains(T obj)—returns true if the obj is in the Queue and returns false otherwise.

   6). boolean empty()—returns true if the Queue is empty, false otherwise

   7). void clear()-empties the Queue.

   Hint: use the head of the list as front and the end of the list as rear. So you need a Node type attribute to remember the last one.

   Use proper indentation and professional documentation.

   To test the methods in your Queue class, write a program that reads input from a file, stores the alphabetical characters (drop the non-alphabetical characters) in a Stack (it follows the "last in, first out" policy) defined in the Java library and your Queue respectively and then check whether the entire string read from the file is a palindrome (it is the same when you read it forward or backward) or not by looking into the characters in the stack and the queue.

   Examples of palindrome: Able was I ere I saw Elba

   Straw? No, too stupid a fad. I put soot on warts.

3. Define a generic OrderedList class by using an array attribute.

   1). The elements in the list should be in ascending order so your add method keeps the order of the list.

   2). When an OrderedList object is created, it can contain at most 100 elements (the length of the array is 100) so the list may be full at any time when an element is added to it. In this case, your add method should create an array of length that is 1.5 times of the length of the existing array and transfer all the elements from the existing array to the new array.

Chapter Four
Recursive Programming

Recursive programming is a technique that breaks up complex computational problems into simpler ones and applies the same technique to each simpler problem repeatedly. The term recursion refers to the fact that the same computation recurs, or occurs repeatedly, as the problem is being solved.

4.1 Basic Static Recursive Methods

A recursive method is one that activates itself in its body. In mathematics, recursive definition is used for complete definition of some formulas and factorial is one of them. Everybody knows that the factorial of an integer is the product of the all integers from one to the integer. For example, 4 factorial is 4*3*2*1. But, how do we write a formula for this definition? A reasonable one would be

n!=n*(n-1)*(n-2)……2*1.

What do the dots mean? Different people may interpret the dots in different ways. To make this definition precise and complete, we may use the following recursive formula:

$$n! = \begin{cases} 1, \; if\ n = 0, \\ n * ((n-1)!), \quad if\ n > 0. \end{cases}$$

How does this formula work? If we want to find 4!, we have to use the second part because 4>0. Thus, we first have 4!=4*(3!). Now, 3! needs the formula again and 3!=3*(2!) because 3>0. Hence, we have 4!=4*3*(2!). Continue working on it, we get 4!=4*3*2*1*(0!). Since 0 is not greater than 0, we have to use the first part of the formula to get 4!=4*3*2*1*1=24.

This formula has two parts: a base part (n=0) that stops the recursion and a recursive part (n>0) that breaks the problem into simpler problems and applies the formula to the simpler problems when needed. Now, we can convert this formula into Java code. Corresponding to applying the same formula to smaller problems in this mathematical definition, we need to apply the same method to smaller problems in Java programming.

```
/**Find the factorial of an integer
  *@param n the integer for finding factorial
  *@return the factorial of n
  */
```

```
public static int factorial(int n)
{
  if(n==0) return 1;     //Base part
  return n*factorial(n-1); //Recursive part
}
```

Again, this method has two parts: base part and recursive part. The recursive part breaks the problem into “smaller” problems and applies the method itself to one or more “smaller” problems. The base part makes sure that the recursive part has a stopping point. An important note is that the recursive part of a recursive method must get “closer” to the base part each time.

A similar problem is to find the nth Fibonacci number. The sequence of the Fibonacci numbers can be listed as following.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ……

Simply speaking, the nth Fibonacci number is the sum of the two Fibonacci numbers right before it. So every Fibonacci number is based on the two Fibonacci numbers right before it. If you think about it carefully, this simple definition has a problem: when does it stop?

$$F(n) = \begin{cases} 0, & if \quad n = 1, \\ 1, & if \quad n = 2, \\ F(n-1) + F(n-2), & if \quad n > 2. \end{cases}$$

The difference of this recursive definition from the factorial one is that the base part of this one has two parts and the recursive part calls the definition itself twice. Because the recursive part goes back to two numbers, the base part needs two numbers to stop it. Here is a Java method corresponding to this definition.

```
  /**Find the nth Fibonacci number
      *@param n the number of the Fibonacci number to be found
      *@return the nth Fibonacci number
      */
public static int Fib(int n)
{
  if (n==1 || n==2 ) return n-1; //Base part
  return Fib(n-1)+Fib(n-2);    //Recursive part
}
```

Both of the above examples are working on numerical computations so they are both value-returning methods. Let’s look at some problems which work on printing out messages. Reversing the digits of a given integer is such a good toy problem.

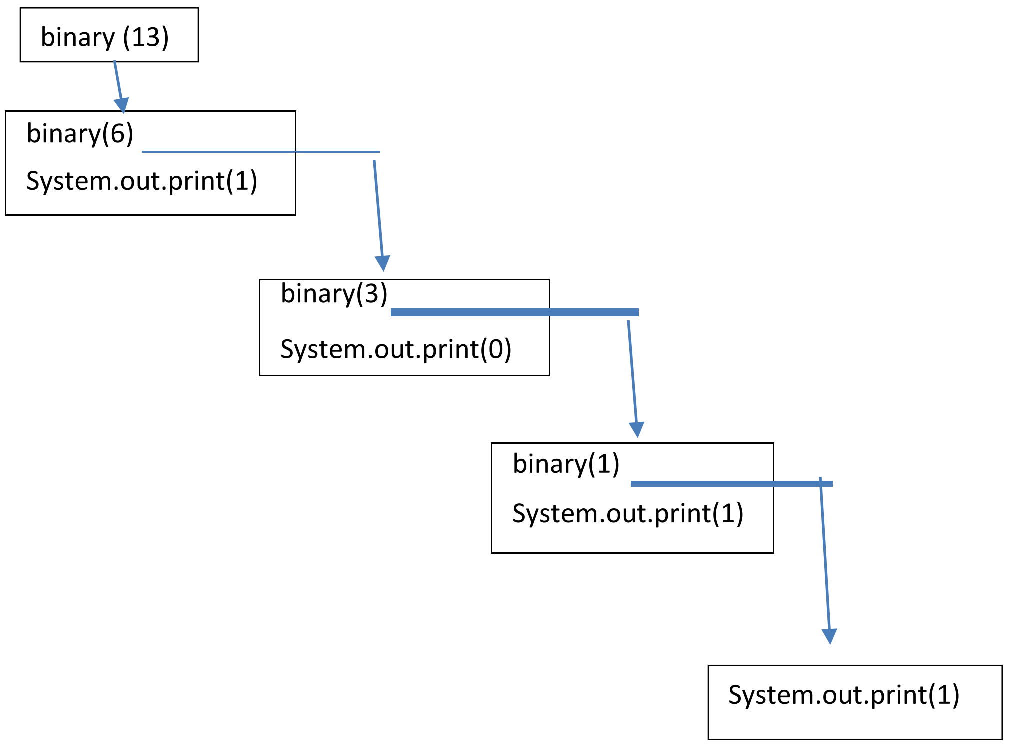
```
/**Print the digits of a given positive integer in reversed order
    *@param n the integer to be printed
```

```
*/
public void reverse(int n)
{
  if(n<10)    //the base part
          System.out.println(n);
   else    //recursive part
   {
     System.out.print(n%10); //print out the last digit
     reverse(n/10);     //apply the method to the rest of the digits
   }
  }
```

The order of the two statements in the recursive part is important. Because we want to reverse the digits, we need to print out the last digit before applying the method to the rest of the digits. To give further illustration, let's look at a similar problem: print out the binary representation of a given positive integer. Here is an algorithm for finding the binary representation of a positive integer: the last digit of the binary representation of a positive integer n is n modulo 2 and the rest contains the binary representation of n/2.

```
/**Print out the binary representation of a given positive integer
   *@param n the positive integer
   */
public void binary(int n)
{
  if(n<2) //base case
      System.out.println(n);
  else //recursive part
  {
   binary(n/2);  //apply the method to n/2
    System.out.print(n%2);   //print out the last digit of the binary
                                         //representation of n
  }
}
```

Again, the order of the two statements in the recursive part is important. The last digit should be printed out after the rest have been printed out so the method should be applied to n/2 before n%2 is printed out so that the rest of the digits are printed out before the last digit is printed out. When a recursive method is executed, the intermediate method calls are stored in the memory as a call stack and executes back when the base case is reached. A recursive method generally looks succinct, but it is hard to follow and debug. Manually drawing the call-stack of a recursive method is the only way to understand how a recursive method works. Let's draw the call stack of the method call binary (13).



The stack is placed upside down here and the plate that contains the method call binary(13) is at the bottom of the stack. When binary(13) is activated, it needs to complete binary(6) before the statement System.out.print(1) is executed. So System.out.print(1) in the second to the bottom plate is placed in the waiting list and it will be executed after binary(6) is completed. To complete binary(6), the statements (binary(3), System.out.print(0)) in the third from the bottom plate should be completed. To complete binary(3), the statements (binary(1), System.out.print(1)) in the fourth from the bottom plate should be completed. Now to complete binary(1), the statement (System.out.print(1)) in the top plate needs to be completed. In the top plate, there is no more recursive call so it can start popping up the plates. Once the statement (System.out.print(1)) in the top plate is executed. A digit 1 is printed out, this top plate is popped out and binary(1) in the second to the top plate is completed. Now the printing statement in the second to the top plate is executed, another digit 1 is printed out and the second to the top plate is popped out. The method call binary(3) in the third to the top plate is completed, the statement following it is executed and a digit 0 is printed out and the third from the top plate is popped out. Now the method call binary(6) in the fourth from the top plate is completed, the printing statement following it is executed, a digit 1 is printed out and the second to the bottom plate is popped out. Now, everything is down, the bottom plate is popped out and the digits printed out are 1101 which is the binary representation of number 13.

4.2 Instance Recursive Methods and Recursive Constructors

A static recursive method is straight forward because a “smaller” problem can be simply passed to the parameter of the method. When we have an instance recursive method, the “smaller” problem may not be displayed through the argument to the parameter. Instead, it is through another object. Let’s look at the factorial problem again: design a simple class that finds the factorial of a non-negative integer.

```
/**A class finding the factorial of a non-negative integer*/
public class Factirial
{
  private int n; //The non-negative integer
  public Factorial(int n){this.n=n;}
  /**Find the factorial of n recursively*/
  public int getFactorial()
  {
   if(n==0) return 1; //base part
      return new Factorial(n-1).getFactorial(); //recursive part
  }
}
```

The difference is that the “problem” is contained in the attribute n and the method getFactorial() here does not have any explicit parameter so the “smaller” problem does not show up through the argument to the parameter. We need to create an object that contains a “smaller” problem and use it to activate the getFactorial method.

It is also possible to apply the recursive technique to constructors of a class. To show this, let’s look at the problem of generating all permutations of a word: design a class that contains a method for finding the permutations of a given word each time when it is activated. Here are all the permutations of the word tim: tim, tmi, itm, imt, mti, mit. In the first pair of permutations, we have the first letter t of the original word attached to all permutations of im. In the second pair, we have the second character i attached to all the permutations of tm. In the third pair, we have the character m attached to all the permutations of ti. Following this example, the recursive algorithm for finding all permutations of a word is clear: get a character from the original word each time and attach it to the front of all the permutations of the new word with this character deleted from the original word. Here is a class that realizes this algorithm.

```
public class PermutationGenerator
{
  private String word; //the original word for permutations
  private int current; //The index of the char used for the first char of the
                              //permutation
  private PermutationGenerator tail; //an object containing the word without
                                                  //the char at current

  /**Explicit constructor
     *@param word the original word
     */
  public PermutationGenerator(String word)
  {
   this.word=word;
   current=0;
   if(word.length()>1
     tail=new PermutationGenerator(word.substring(1));
  }
  /**Compute the next permutation of the word based on the value of
    *current and the previous permutation
     *@return the next permutation
     */
  public String nextPermutation()
  {
   if(word.length()==1) //base part
   {
     current++;
     return word;
  }
  String r=word.charAt(current)+tail.nextPermutation(); //generate the
                                                        //permutation recursively
  /*Update current and the tail object for next permutation if all
     permutations from this tail have been generated
  */
  if(tail.current==(tail.word).length())
  {
   current++;
   if(current<word.length())
      tail=new PermutationGenerator(word.substring(0,
                                    current)+word.substring(current+1));
  }
  return r;
}

/**Get the value of the index of the char used as the first char of the
   *permutation
  *@return the index of the char used as the first char of the permutation
  */
 public int getCurrent(){return current;}
}
```

To use this class, one needs to remember that the getPermutation method generates a permutation each time when it is activated and updates the current position of the character and tail object when needed. So one needs a loop to generate all permutations of a word. For example, the following section of code generates all permutations of the word "time".

```
String string=”time”;
PermutationGenerator generator=new PermutationGenerator(string);
for(int i=1; generator.getCurrent()<string.length(); i++)
                System.out.println(i+ “:  “+generator.nextPermutation());
```

Another way to solve the permutation problem is to use a loop in the method and store all permutations generated in a list.

```
public class PermutationGenerator
{
 private String word;
 private ArayList<String> permutationList; //the list containing all
                                                         //permutations of this word
 public PermutationGenerator(String word)
 {
   this.word=word;
  permutationList=new ArrrayList<>();
 }
 public void getPermutations()
 {
   if(word.length()==1) //base part when the word has only one character
   {
     permutationList.add(word);
     return;
   }
  for(int i=0; i<word.length(); i++)//Combine a loop with a recursive call
  {
   PermutationGenerator generator
           =new PermutationGenerator(word.substring(0,i)
                                                     +word.substring(i+1));
   generator.getPermutations(); //Recursive call to generate all
                                          //permutations of the subword
   for(String s:generator.permutationList)
      permutationList.add(word.chatAt(i)+s);//attach the character at i to
                                                       //each of the permutations of the
                                                         //subword
 }

 public ArrayList<String> getList(){return permutationList;}
}
```

4.3 Recursion with Backtracking

In solving a problem using recursion, there may be some situations where different ways lead from different positions and some of them don't lead to a solution. After trying one path unsuccessfully or having processed a solution when a result is reached, we need to return back to the crossroad and try to find a solution using another path. However, we need to make sure that such a return is possible and all paths can be tried. This technique is called backtracking and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. There are several typical examples that need backtracking recursive technique: eight-queen problem, maze-path problem and the Sudoku problem. We work out the eight-queen problem here and leave the other two to the students as programming projects.

Eight-queen problem is stated as the following: place eight queens on a chessboard in such a way that no queen can attack any other. The rules of chess say that a queen can take another piece if it lies on the same row, on the same column, or on the same diagonal line as the queen. Here is our pseudo-code for a putQueen method.

```
putQueen(row)
    for every position col in the same row
        if position (row, col) is available
             place the next queen at (row, col)
             if there are more rows
                  putQueen(next row)
             else process the successful result
             remove the queen at position (row, col)
```

The key steps for backtracking are the loop and removing the queen at position (row, col). When the removing queen step is reached, there are two possibilities: a dead-end or a success is reached. In either case, we need to remove the current queen so that another way can be tried. The loop provides alternative ways when one trial reaches a dead-end or a success. Although it does not need to be a loop, there needs to be a way for giving alternative trials. In a traditional maze problem, it certainly cannot have a loop. Instead, it needs to try all the four different directions: up, down, left and right so when one direction reaches a dead-end or a success, other directions should be tried. Think about how to track back: go back one step and change direction. How to go back one step? Erase what you have done in this step. How to change direction? Another choice (another iteration in a loop or simply a branch statement given by an if-else statement) is switched to.

```
/*A class containing a method that finds all possibilities for placing eight
queens on a regular chess board, a value 0 indicates a position is open
and a value 1 indicates that the position has a queen*/
public class EightQueens
{
   private int[][] board;
   private static int dim=8;

   public EightQueens(){board=new int[dim][dim]; }

   /**Check whether a row is available or not for placing a queen
       *@param row the row being checked
       *@return false if the row contains a queen, true otherwise
       */
  private boolean rowAvailable(int row)
  {
     for(int j=0; j<dim; j++
     {
        if(board[row][j]==1)
           return false;
     }
     return true;
 }

    /**Check whether a column is available or not for placing a queen
       *@param col the column being checked
        *@return false if the column contains a queen, true otherwise
        */
   private boolean columAvailable(int col)
   {
      for(int j=0; j<dim; j++)
      {
        if(board[j][col]==1)
           return false;
     }
     return true;
 }

    /**Check whether the backward diagonal line containing position
      *(row, col) is available for placing a queen
      *@param row the row index of the position
      *@param col the column index of the position
      *@return false if the backward diagonal line contains a queen,
      *              true otherwise
      */
   private boolean backwordDiagAvailable(int row, int col)
   {
      for(int i=row, j=col; i<dim && j<dim; i++, j++) //check the
                                                       //positions below (row, col)
      {
        if(board[i][j]==1)
           return false;
       }
      for(int i=row-1, j=col-1; i>=0 && j>=0; i--, j--)//check the positions
                                                          //above (row, col)
      {
        if(board[i][j]==1)
           return false;
      }
     return true;
 }

 /**Check whether the forward diagonal line containing position (row,
    *col) is available for placing a  queen
    *@param row the row index of the position
    *@param col the column index of the position
  *@return false if the forward diagonal line contains a queen, true
   *             otherwise
   */
   private boolean forwardDiagAvailable(int row, int col)
   {
      for(int i=row, j=col; i>=0 && j<dim; i--, j++) //check the positions
                                                      //above (row, col)
      {
        if(board[i][j]==1)
           return false;
       }
    for(int i=row+1, j=col-1; i<dim && j>=0; i++, j--)//check the
                                               //positions above (row, col)
     {
        if(board[i][j]==1)
           return false;
      }
     return true;
 }

 public void putQueen(int row)
 {
   for(int j=0; j<dim; j++)
   {
      if(rowAvailable(row) && columnAvailable(j)
           && backwardDiagAvailable(row, j)
           &&forwardDiagAvailable(row, j))
      {
           board[row][j]=1;
           if(row<dim-1)
                putQueen(row+1);
           else
```

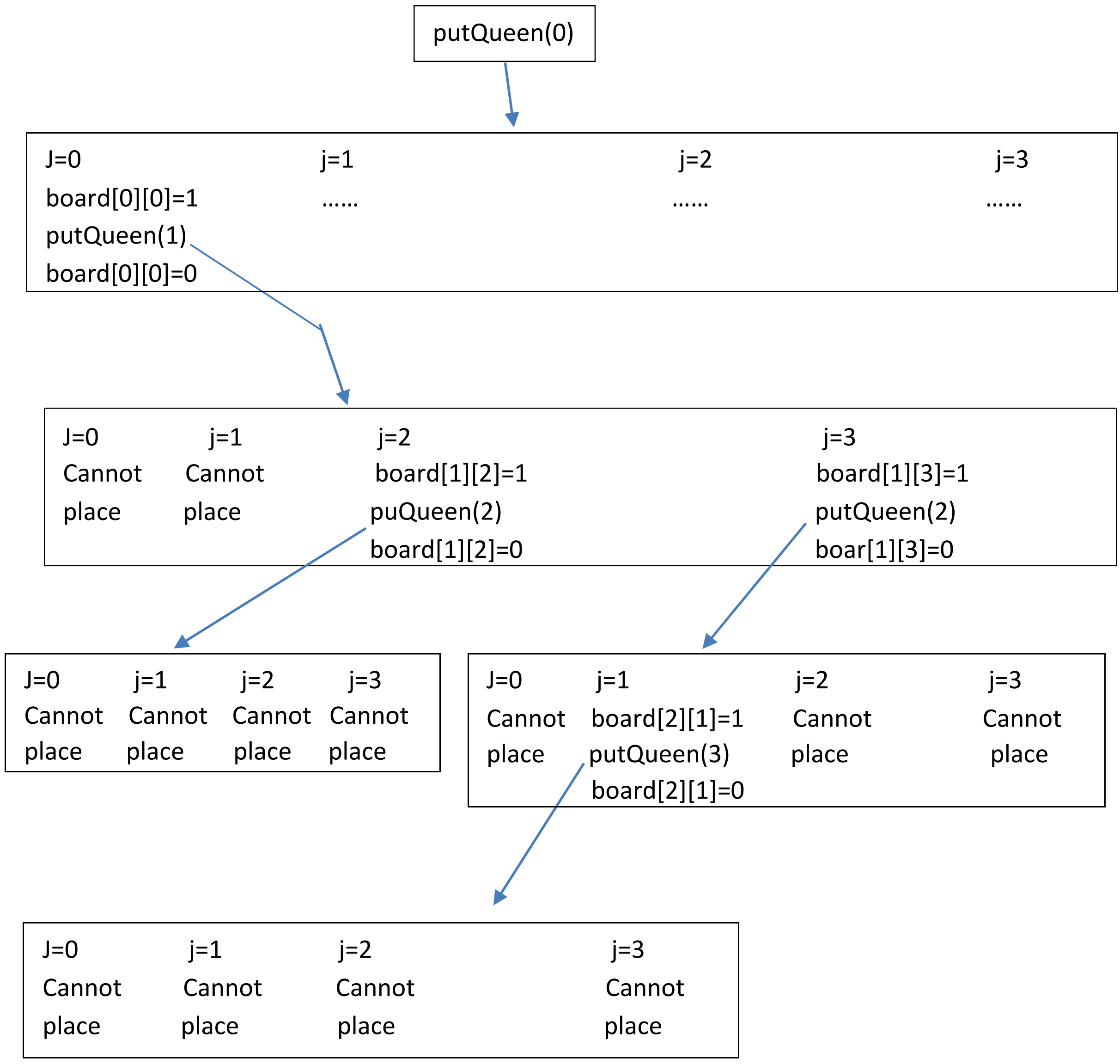
```
                  {
                     printBoard(); //success
                     System.out.println("******************");
                  }
                  board[row][j]=0; //remove queen for backtracking
               }
             }
          }
        public void printBoard()
        {
          for(int i=0; i<dim; i++)
          {
              for(int j=0; j<dim; j++)
                 if(board[i][j]==0)
                    System.out.print("X");
                 else
                     System.out.print("Q")
            System.out.println();
          }
      }

      public void findSolutions()
      {
        putQueen(0);
      }
}
```

The regular eight-queen problem has 92 solutions so it is hard to manually follow through the entire process. To understand how this program actually works, it is better to change the dimensions of the board to a fake 4x4 board which has only two solutions:

0	1	0	0
0	0	0	1
1	0	0	0
0	0	1	0

0	0	1	0
1	0	0	0
0	0	0	1
0	1	0	0

Let's try to draw part of a call-stack following the code for a four-queen problem.



In this part of our call stack, we only draw the case for the first cell in the first row of our 4x4 board and have left the cases of the second, third and fourth cells of the first row to the students. Right now, the board contains all 0s:

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Let's see how it works. Again, we are drawing the stack upside down and the top one in our graph actually should be the plate in the bottom of the stack. When j=0 in the second plate to the bottom one, the cell at (0, 0) is available so a digit 1 is placed into this position, then putQueen(1) is activated and the digit at (0, 0) is changed back to 0 after putQueen(1) is completed. To complete putQueen(1), it needs to go through every cell in the row with index 1 so j needs to go through 0, 1, 2, and 3. Right now, the board contains a digit 1 at position (0, 0) and all others are 0:

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

In the third plate from the bottom one, we can notice that both (1, 0) and (1, 1) are not available. When j=2 and j=3, we have the stack split out into two branches. Let's follow through the case when j=2 first. In this case, it places a digit 1 to the cell at position (1, 2), activates putQueen(2), and then changes the digit 1 at position (1, 2) back to 0 after the method call putQueen(2) is completed. Now, the board contains digit 1 at positions (0, 0) and (1, 2):

1	0	0	0
0	0	1	0
0	0	0	0
0	0	0	0

To complete the method call putQueen(2), we need to go to the fourth level of the stack in which none of the cells in the row with index 2 is available so it pops back and the digit 1 at position (1, 2) is changed back to 0:

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Now, the flow of execution in level three of the stack moves to j=3 which places a digit 1 to the cell at position (1, 3), calls the method putQueen(2) and changes the digit back to 0 after the method call is completed. Now, the board contains two 1s again:

1	0	0	0
0	0	0	1
0	0	0	0
0	0	0	0

To complete putQueen(2), the flow moves to the right plate in the fourth level of the stack. In this plate, cell (2, 0) is not available so it moves to j=1. The cell at (2, 1) is an available one so a digit 1 is paced to the cell (2, 1), then putQueen(3) is activated and cell at (2, 1) is changed back to 0 after this method call is completed. Now, the board contains three 1s:

1	0	0	0
0	0	0	1
0	1	0	0
0	0	0	0

To complete putQueen(3), the flow needs to first move to the plate in the fifth level of the stack in which none of the cells in the row with index 3 is available so this plate is popped out and the digit in the cell at (2, 1) is changed back to 0. The board goes back to two 1s now:

1	0	0	0
0	0	0	1
0	0	0	0
0	0	0	0

Now, the flow of execution moves to j=2 and j=3 in the right plate of the fourth level and neither of the cells with these two column indices in the row of index 2 is available. Thus, this plate is popped out, putQueen(2) is completed, the statement board[1][3]=0 is executed and the left plate in the third level of the stack is popped out. The board now goes back to one 1:

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Since putQueen(1) has been completed now, the statement board[0][0]=0 following it is executed and the flow of execution moves to the case for j=1 in the plate of the second level and the second cell of first row is being tried. As expected, the board goes back to the original one without any digit 1 in it.

Review Questions

1. What is recursion?
2. What are the two necessary parts (cases) for a recursive method?
3. What is the base part of a recursive method for?
4. What is the recursive part of a recursive method for?
5. Why are many recursive methods very time inefficient?
6. When a recursive call of a static method is applied to a "smaller" problem, its "smallness" shows up in the argument passed to its parameter. Where does the "smallness" show up in an instance recursive method?
7. Write a static recursive method that finds and returns the sum of all integers from 1 through a given integer n.
8. Design a class with an instance method that finds and returns the sum of all integers from 1 through a given integer n.
9. Inspect the following method, draw a stack of the method calls, and then describe how the method call gcd(120, 75) works.
   ```
   public static int gcd(int a, int b)
   {
     if(b==0) return a;
     return gcd(b, a%b);
    }
   ```
10. What condition should the parameter i in the following method satisfy so that the method could work? What does the method call print(a, n) do if a is an int type array and n is an integer satisfying the condition required by the parameter i.
    ```
    public static void print(int [] a, int i)
     {
      if(i==-1)
         return;
      else
      {
        print(a, i-1);
        System.out.print(a[i]+ " ");
      }
    }
    ```
11. Implement a GCD class that has two int type attributes, an explicit constructor and an instance RECURSIVE method getGCD() that is an instance version of the

static method in problem #9 and finds the greatest common divisor of the integers of the object when it is called by a GCD object.

12. What is backtracking?
13. Complete the recursive call stack for the queens problem with a 4x4 board.

Programming Projects.

1. Consider the following algorithm for computing x^n for an integer n. First, you know $x^0 = 1$. If $n < 0,\ then\ x^n = \frac{1}{x^{-n}}$. If n is positive and even, then, $x^n = (x^{\frac{n}{2}})^2$. If n is positive and odd, then $x^n = x^{n-1} * x$.Implement a static recursive method that uses this algorithm.
2. The Hanoi game is a one person game. In this game, there are three pegs and n disks of different sizes. Before the game is started, all n disks are on one peg and each disk is smaller than the ones below it. The goal is to move all n disks from this peg to another peg by using the third peg as help without ever putting one disk over one that is smaller. Design a class with an instance recursive method that prints out how each disk should be moved for achieving the goal.
3. Design a static generic binary search method that searches through an array of ascending values for a given value. If the value is in the array, the index of the first value in the array is returned. Otherwise, -1 is returned. To indicate which section of the array is being searched through, a begin index parameter and an end index parameter are needed. When the method is called, 0 is passed to the begin index parameter and length-1 is passed to the end index parameter.

4. Here is an ancient algorithm for approximating the square root of a given non-negative number: first guess a number for the square root of the given number and then a better guess is the average of this guess and the quotient of the original number divided by this guess. For example, we want to approximate the square root of 250 in the following table by using 10 as our first guess.

The first guess	10
The second guess	(10+250/10)/2=17.5
The third guess	(17.5+250/17.5)/2=15.8928
The fourth guess	(15.8928+250/15.8928)/2=15.8116
The fifth guess	(15.8116+250/15.81160)/2=15.8114

Since the fourth guess and the fifth guess have the first three decimal places equal, we believe that we have approximated the square t of 250 accurate to three decimal places. Actually, the square root of 250 is 15.811388300.
Write a static recursive method that approximates the square root of a given number to a certain given accuracy.

5. Design a class with a recursive instance method that uses the ancient algorithm given in the previous problem to approximate the square root of a given number to a certain given accuracy.
6. A word is elf-ish if it contains the letters e, l and f in it. Write a recursive method that checks whether a given word is elf-ish or not.
7. A word is X-ish if it is contained in another word. Define a recursive method X-ish that has two String parameters and checks whether the first one is contained in the second one or not.
8. Design a class with a method that uses recursion to find all subsets (subwords) of a set of characters (a word). For example, all subsets of the character set **word** include word, ord, wrd, wod, wor, rd, od, or, wd, wr, wo, w, o, r, d, and {}. A set with n characters has 2^n subsets including the empty subset and itself.
9. A maze is implemented as a two-dimensional character array in which passages are marked with 0s, walls by 1s, exit position by the letter e and the initial position by the letter b. For example,

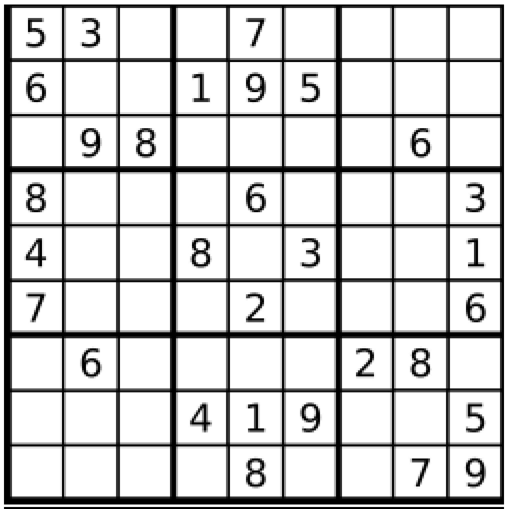
   ```
   1 1 1 1 1
   b 0 1 0 1
   1 0 1 0 e
   0 0 0 0 1
   1 1 1 1 1
   ```

   is a maze that has only one solution.
   Write a program that asks the user to enter the dimensions of a maze and the maze, finds and prints out **graphical mazes** of all possible paths (a maze is printed for each path, each maze entered may have multiple paths to get out) by putting p in each cell of the path. The above maze has only one solution printed out to its right.

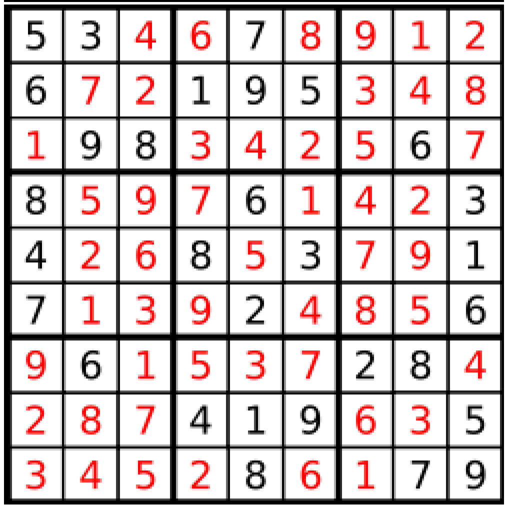
   A sample maze you may use is

   ```
   11111111111
   10000010001
   10100010101
   e0100000101
   10111110101
   10101000101
   10001010001
   11111010001
   101b1010001
   10000010001
   11111111111
   ```
10. Sudoku puzzle is a logic-based number placement puzzle. The objective is to fill a 9X9 grid with digits so that each row, each column, and each of the nine 3X3

sub-grids that compose the grid contains all of the digits from 1 to 9 (so no digit repetition in each row, each column, and each of the nine 3X3 sub-grids). The puzzle setter provides a partially completed grid, of course. Design a class with a recursive method that can complete any Sudoku puzzle provided. The following is an example. The first one is a puzzle and the second one is the result after completed.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Chapter Five
Multiple Threading

A thread is a program unit that is executed independently of other parts of the program. Multiple threading means that two or more threads are running at the same time. Multiple images of webpage at the same time and animation to show moving figures are among the popular multiple threading examples.

The Java virtual machine allows for multi-threaded programs to be written in Java. Unless multiple processors are used, multiple threads don't actually run concurrently. When multiple threads are executed in a single processor, the processor allocates little time slots to each thread unit and gives the users an illusion that threads are running concurrently.

5.1 The Thread Class in Java Library

The Java library has a class named Thread that provides the building block for Java programmers to build their own threads. The most important methods in the Thread class are the start method and the run method. The start method causes the thread to begin execution and the Java Virtual machine calls the run method of this thread. Thus the run method should contain the code that dictates how the thread behaves. This is realized by two ways. One way is to design a class that extends the Thread class and overrides the run method. Another way is to design a class that implements the Runnable interface and its run method and pass an object of this class to a constructor of the Thread class. When a thread is constructed using a separate Runnable object, the activation of the thread object calls the Runnable object's run method.

5.2 Extending the Tread Class

In this section, we use an example to illustrate how to design a thread by extending the Thread class in the library. We first look at some constructors and methods of the Thread class.

Thread(String name)—constructs a Thread object with the given name.
String getName()—returns the thread's name.
static void sleep(long millis)—causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
void start()—causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

Now, let’s design a thread class that sends out a greeting message 10 times.

```
public class GreetingThread extends Thread
{
  private String greeting; //the message
  private int time; //sleeping time
   /**Constructor*/
  public GreetingThread(String name, String aGreeting, int time)
  {
   super(name);
   greeting=aGreeting;
   this.time=time;
  }
  /**Override the run method of the Thread class*/
  public void run()
  {
   try{
        for(int i=1; i<=10; i++)
        {
            Date now=new Date();
            System.out.println(now+ “, ”+getName()+ “ says ”+greeting+ “.”);
            sleep(time);
         }
      }catch(InterruptedException e){return;}
   }
 }
}
```

Now, we can test our GreetingThread class.

```
public class GreetingThreadTest
{
  public static void main(String[] args)
  {
   GreetingThread t1=new GreetingThread(“Mike”, “Hello, world!”, 1000);
   GreetingThread t2=new GreetingThread(“Bob”, “Goodbye, world!”, 500);
   t1.start();
   t2.start();
   }
  }
```

It is important to understand that the Java Virtual Machine schedules this thread to run concurrently with the other threads only when it activates the start method. If a thread object directly activates its run method without going through the start method, it behaves as a regular class. More precisely, the code

```
   GreetingThread t1=new GreetingThread(“Mike”, “Hello, world!”, 1000);
   GreetingThread t2=new GreetingThread(“Bob”, “Goodbye, world!”, 500);
   t1.run();
   t2.run();
```

would send out the message

```
          current time, Mike says Hello, world
```

10 times and then send out the message

```
           current time, Bob says Goodbye, world
```

10 times.

5.3 Implementing the Runnable Interface

Another way to construct a thread is to pass an object of a class that implements the Runnable interface in the library into one of the following Thread constructors. In this case, the task of the thread is placed in the run() method of the Runnable interface.

Thread(Runnable target)—Construct a new Thread object.
Thread(Runnable target, String name)—Construct a new Thread object with the given name.

Let’s rewrite the thread in the previous section using the Runnable interface.

```
public class Greeting implements Runnable
{
  private String greeting;
  private String name;
  private int time; //sleeping time in milliseconds

 public Greeting(String greeting, String name, int time)
 {
   this.greeting=greeting;
   this.name=name;
   this.time=time;
  }
 public void run()
 {
   try{
            for(int i=1; i<=10; i++)
            {
              Date now=new Date();
              System.out.println(now+ “, ”+name+ “ says ”+greeting+ “.”);
              Thread.sleep(time);
             }
        }catch(InterruptedException e){return;}
  }
}

public class RunnableTest
{
  public static void main(String[] args)
  {
     Greeting g1=new Greeting(“Hello, world!”, “Mike”, 1000);
     Thread t1=new Thread(g1);
     Greeting g2=new Greeting(“Goodbye, world!”, “Bob”, 500);
     Thread t2=new Thread(g2);
     t1.start();
     t2.start();
   }
}
```

One difference between extending the Thread class and implementing the Runnable interface is that by extending Thread, each of your threads has a unique object associated with it, whereas through implementing Runnable, many threads may share the same object.

A class that implements a Runnable is not a thread. For a Runnable to become a thread, you have to create an object of Thread and pass your Runnable object into it as a target. Thus, it is not scheduled as a thread when a Runnable object activates its run method directly. More precisely, the code

```
   Greeting g1=new Greeting( “Hello, world!”, “Mike”, 1000);
   Greeting g2=new Greeting( “Goodbye, world!”, “Bob”, 500);
   g1.run();
   g2.run();
```

would send out the message

```
            current time, Mike says Hello, world
```

10 times and then send out the message

current time, Bob says Goodbye, world

10 times.

So in most cases, the Runnable interface should be used when you are only planning to override the run method and not any other thread methods. Classes should not be subclassed unless the programmer intends to modifying or enhancing the fundamental behavior of the classes.

When there is a need to extend a superclass other that the Thread class, it may be more appropriate to implement the Runnable interface than extend the Thread class because a Java class can extend only one class and we can extend the other class while implementing the Runnable interface. Another advantage of using the Runnable interface is that multiple Thread objects may share the same Runnable object so it saves memory and code/

5.4 The Interrupt Method and Interrupted Exception

When a thread is on the way of another more important thread, it needs to stop and give its way to the other one. But, it may cause some problems if a thread is stopped abruptly. That is, a thread needs to stop at an appropriate point when it needs to be stopped. The following three methods serve this purpose.

void interrupt—Notifies the thread that it should clean up and terminates.

static boolean interrupted()—Tests whether the current thread has been interrupted.

boolean isInterrupted()—Tests whether this thread has been interrupted.

It is important to point out that the interrupted() and isInterrupted() methods are different although both are used to test whether a thread has been interrupted. The interrupted() method is a static method in Thread class and the interrupted status of the thread is cleared by this method. Therefore, if a thread was interrupted, calling interrupted() once would return true, while a second call to it would return false until the current thread is interrupted again. The isInterrupted() method is an instance method that tests whether this thread instance has been interrupted and the interrupted stastus of the thread is unaffected by this method.

A thread does not have to terminate when it is interrupted. Interrupting is only a general mechanism for getting the thread’s attention, even when it is sleeping. If a thread is sleeping, it cannot execute code that checks for interruption. The sleep() method throws an InterruptedException whenever a sleeping method is interrupted. You need to catch the exception and terminates the thread if this occurs.

5.5 Racing Conditions and Deadlocks

Each thread may have its own local variables, but all threads may share access to an instance field of an object. This creates a problem of race condition because all threads, in their race to complete their respective tasks, manipulate a shared field and the end result depends on which of them happens to win the race. Let's use the following example to illustrate this situation.

```
/**Define a simple BankAccount class that does not generate interest*/
public class BankAccount
{
  private double balance;

  public BankAccount(double amount)
 {
   balance=amount;
  }
  /**Deposit some amount to the account
     *@param amount the amount of money deposited into this account
     */
  public void deposit(double amount)
  {
   System.out.print("Depositing "+amount);
   double newBalance=balance+amount;
    //Place some garbage code here to pretend that depositing takes some
    //time
   in sum=0;
   for (int i=0; i<10000; i++)
       sum+=I;
   System.out.println(", new balance is "+newBalance);
    balance=newBalance;
  }
    /**Withdraw some amount from the account
     *@param amount the amount of money withdrawn from the account
     */
  public void withdraw(double amount)
  {
   System.out.print("Withdrawiting "+amount);
   double newBalance=balance-amount;
    //Place some garbage code here to pretend that withdrawing takes
```

```
                    //some time
   in sum=0;
   for (int i=0; i<10000; i++)
       sum+=I;
   System..out.println(", new balance is "+newBalance);
    balance=newBalance;
  }

  public getBalance(){return balance;}
}
```

Now, we may design two threads, one deposit thread and one withdraw thread.

```
public class DepositThread extends Thread
{
  private BankAccount account;
  private double amount;

  public DepositThread(BankAccount account, double amount)
  {
   this.account=account;
   this.amount=amount;
   }

  /**This thread deposits the given amount of money to the account ten
      times*/
   public void run()
   {
     try
     {
        for(int i=1; i<=10 && !isInterrupted(); i++)
        {
          account.deposit(amount);
          sleep(500);
        }
      }catch(InterruptedException e){return;}
    }
}

public class WithdrawThread extends Thread
```

```
{
  private BankAccount account;
  private double amount;

  public WithdrawThread(BankAccount account, double amount)
  {
   this.account=account;
   this.amount=amount;
   }

  /**This thread withdraw the given amount of money from the account ten
      times*/
   public void run()
   {
     try
     {
         for(int i=1; i<=10 && !isInterrupted(); i++)
        {
          account.withdraw(amount);
          sleep(500);
        }
      }catch(InterruptedException e){return;}
    }
}
```

Now we may test our threads and see how a racing condition happens.

```
public class BankAccountThreads
{
  public static void main(String[] a)
  {
     BankAccount account=new BankAccount(0);
   DepositThread t1=new DepositThread(account, 100);
   WithdrawThread t2=new WithdrawThread(account 100);

    t1.start();
    t2.start();
   }
  }
```

When you execute the code, you may get the following output due to the race condition.

```
Withdrawing 100.0Depositing 100.0, new balance is -100.
, new balance is 100
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
```

What has happened? The withdraw thread was scheduled first so the withdraw method was called the first time, the message “withdrawing 100.0” was printed out and the local variable new Balance was calculated. But, after the first two statements of the withdraw method were executed, the processor was assigned to the deposit thread so the first two statements of the deposit method were executed. Therefore, the message “Depositing 100.0” was printed out and the local variable newBalance was assigned to 100.0. At this moment, the processor was given back to the withdraw thread, the rest of the withdraw method was executed and the message “, new balance is -100.0” was sent out. Then, the processor was assigned back to the deposit thread, the rest of the deposit method was executed and “, new balance is 100.0” was sent out. It needs to point out that the last balance sent out is the value of the local variable, not the actual balance of the account. The actual balance of the account is 0.0.

This problem occurred because the deposit thread grabbed the processor and modified the state of the account balance before the withdraw thread completed the withdraw process. To solve this problem, we need a thread to “lock” an object that it is working on so other thread cannot interfere it until it is complete. There are two ways to do this. One way is to use the keyword synchronized in the header of a method or a block of code of a method. To illustrate the first way, we may modify the deposit and withdraw method in the BankAccount to the following.

……

```
/**Deposit some amount to the account
    *@param amount the amount of money deposited into the account
    */
  public synchronized void deposit(double amount)
  {
    System.out.print(“Depositing ”+amount);
    double newBalance=balance+amount;
     //Place some garbage code here to pretend that depositing takes some
      //time
    in sum=0;
    for (int i=0; i<10000; i++)
        sum+=I;
    System..out.println(“, new balance is ”+newBalance);
     balance=newBalance;
  }
   /**Withdraw some amount from the account
    *@param amount the amount of money withdrawn from the account
    */
  public synchronized void withdraw(double amount)
  {
    System.out.print(“Withdrawiting ”+amount);
    double newBalance=balance-amount;
     //Place some garbage code here to pretend that withdrawing takes
     //some time
    in sum=0;
    for (int i=0; i<10000; i++)
        sum+=I;
    System.out.println(“, new balance is ”+newBalance);
     balance=newBalance;
  }
```

The synchronized keyword can also be applied to a block of code. For example, the methods can be modified in the following way.

```
/**Deposit some amount to the account
   *@param amount the amount of money deposited into the account
   */
 public void deposit(double amount)
 {
  synchronized(this)
  {
     System.out.print(“Depositing ”+amount);
     double newBalance=balance+amount;
      //Place some garbage code here to pretend that depositing takes
      //some time
     in sum=0;
     for (int i=0; i<10000; i++)
         sum+=I;
     System..out.println(“, new balance is ”+newBalance);
      balance=newBalance;
  }
 }
   /**Withdraw some amount from the account
    *@param amount the amount of money withdrawn from the account
    */
 public void withdraw(double amount)
 {
  synchronized(this)
  {
     System.out.print(“Withdrawiting ”+amount);
     double newBalance=balance-amount;
      //Place some garbage code here to pretend that withdrawing takes
      //some time
     in sum=0;
     for (int i=0; i<10000; i++)
         sum+=I;
     System..out.println(“, new balance is ”+newBalance);
      balance=newBalance;
    }
 }
```

After the keyword synchronized is applied to the header of these methods and when one of them is activated by an object in a thread, the thread locks the object so that no other thread can access the object until it is released. When an object is locked by a thread, we say that the thread owns the lock. If another thread tries to access the object, that thread would be temporarily deactivated. The thread scheduler periodically reactivates such a thread so that it can again try to access the object. If the object is still locked, the thread is again deactivated. Eventually, the waiting thread can get access to the object when the other thread has unlocked the object. Now, we would have the following output if we execute the code.

```
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0
Withdrawing 100.0, new balance is -100.0
```

Now, we need to take care of another problem: some banks may not allow negative balance. So when the withdraw amount is less than the balance, we need to block the withdraw process and wait for a deposit so that the balance is enough for the withdraw amount. One choice is to modify the withdraw method in the following way.

```
   /**Withdraw some amount from the account
    *@param amount the amount of money withdrawn from the account
    */
  public synchronized void withdraw(double amount)
  {
        while(balance<amount); //wait for somebody to deposit enough so
                                           //balance >=amount
       System.out.print(“Withdrawiting ”+amount);
       double newBalance=balance-amount;
        //Place some garbage code here to pretend that withdrawing takes
        //some time
       in sum=0;
```

```
      for (int i=0; i<10000; i++)
          sum+=I;
      System..out.println(", new balance is "+newBalance);
       balance=newBalance;
 }
```

This modification may cause a serious problem: deadlock. When the withdraw method is activated, the object activating it is locked by the withdraw thread so this object cannot be accessed by the deposit thread. But, without being modified by the deposit thread, the balance field of the object cannot be increased to satisfy the requirement balance>=amount. It may happen that one thread locks an object, but waits for another thread to do some essential work to this object. If that other thread is currently waiting to lock the same object, then neither of the two threads can proceed. This situation is called a deadlock.
The following Thread methods can be used to resolve the deadlock problem.

public void wait()—Causes the current thread to releases an object lock and gets blocked until another thread invokes the notify() method or the notifyAll() method. Throws an InterruptedException if the current thread is interrupted while it is waiting for notification.
public void notify()—Unblocks this thread.
public void notifyAll()—Unblocks all threads that are blocked.

The wait() method temporarily releases an object lock and blocks the current thread. It does not simply deactivate a thread in the same way as a thread that reaches the end of its time slice. It is in a blocked state and cannot be activated by the thread scheduler until it is unblocked by the notify() method or notifyAll() method.
Now we get the final modification of our deposit and withdraw methods.

```
 /**Deposit some amount to the account
     *@param amount the amount of money deposited into the account
     */
  public synchronized void deposit(double amount)
  {
    System.out.print("Depositing "+amount);
    double newBalance=balance+amount;
     //Place some garbage code here to pretend that depositing takes some
     //time
    in sum=0;
    for (int i=0; i<10000; i++)
        sum+=I;
  System..out.println(“, new balance is ”+newBalance);
   balance=newBalance;
   notifyAll();
 }
   /**Withdraw some amount from the account
    *@param amount the amount of money withdrawn from the account
    */
 public synchronized void withdraw(double amount) throws
                                                    InterruptedException
 {
  while(balance<amount)
        wait();
  System.out.print(“Withdrawiting ”+amount);
  double newBalance=balance-amount;
   //Place some garbage code here to pretend that withdrawing takes
   //some time
  in sum=0;
  for (int i=0; i<10000; i++)
     sum+=I;
  System.out.println(“, new balance is ”+newBalance);
   balance=newBalance;
 }
```

Now, we look at the second way of synchronizing the access of an object. In this way, we use the lock() and unlock() methods of the ReentrantLock class to surround a block of code and lock an object. A potential problem of using this way is that the call to unlock() would never happen if the code between the calls lock() and unlock() throws an exception. After an exception is thrown, the current thread continues to hold the lock and no other thread can do anything.

To overcome this problem, we need to place the call to unlock into a finally clause. Corresponding to the wait(), notify() and notifyAll() methods used above, we have await() and signalAll() methods from the Condition class. Let’s modify our threads for using the lock() and unlock() methods.

```
/**Define a simple BankAccount class that does not generate interest*/
public class BankAccount
{
  private double balance;
  private Lock balanceChangeLock;
  private Condition sufficientFundsCondition;
```

```
 public BankAccount(double amount)
{
  balance=amount;
  balanceChangeLock=new ReentrantLock();
  sufficientFundsCondition=balanceChangeLock.newCondition();
 }
 /**Deposit some amount to the account
    *@param amount the amount of money deposited into the account
    */
 public void deposit(double amount)
 {
  balanceChangeLock.lock();
  try
  {
     System.out.print("Depositing "+amount);
     double newBalance=balance+amount;
      //Place some garbage code here to pretend that depositing takes
      //some time
     in sum=0;
     for (int i=0; i<10000; i++)
         sum+=I;
     System..out.println(", new balance is "+newBalance);
     balance=newBalance;
     sufficientFundCondition.signallAll();
    }finally{balanceChangelock.unlock();}
 }
   /**Withdraw some amount from the account
    *@param amount the amount of money withdrawn from the account
    */
 public void withdraw(double amount)
 {
   balanceChangeLock.lock();
  try
  {
      while(balance<amount)
          sufficientFundsCondition.await();
     System.out.print("Withdrawiting "+amount);
     double newBalance=balance-amount;
      //Place some garbage code here to pretend that withdrawing takes
```

```
        //some time
       in sum=0;
       for (int i=0; i<10000; i++)
           sum+=I;
       System..out.println(“, new balance is ”+newBalance);
        balance=newBalance;
      }fianally{balanceChangeLock.unlock();
  }

  public getBalance(){return balance;}
}
```

Now, we may design two threads, one deposit thread and one withdraw thread.

```
public class DepositThread extends Thread
{
  private BankAccount account;
  private double amount;

  public DepositThread(BankAccount account, double amount)
  {
    this.account=account;
    this.amount=amount;
   }

  /**This thread withdrawthe given amount of money ten times*/
   public void run()
   {
     try
     {
        for(int i=1; i<=10 && !isInterrupted(); i++)
        {
          account.deposit(amount);
          sleep(500);
        }
      }catch(InterruptedException e){return;}
    }
}

public class WithdrawThread extends Thread
```

```
{
  private BankAccount account;
  private double amount;

  public WithdrawThread(BankAccount account, double amount)
  {
   this.account=account;
   this.amount=amount;
   }
   public void run()
   {
     try
     {
         for(int i=1; i<=10 && !isInterrupted(); i++)
        {
          account.withdraw(amount);
          sleep(500);
        }
      }catch(InterruptedException e){return;}
    }
}
```

Now we may test our threads and see how racing condition happens.

```
public class BankAccountThreads
{
  public static void main(String[] a)
  {
     BankAccount account=new BankAccount(0);
   DepositThread t1=new DepositThread(account, 100);
   WithdrawThread t2=new WithdrawThread(account 100);

    t1.start();
    t2.start();
   }
  }
```

Review Questions

1. What is a thread?
2. What are the two ways to create a thread in Java?
3. How do multiple threads work with only one processor?

4. If GreetingRunnable is a class that implements the Runnable interface and its method **run()** prints out the string supplied to the constructor three times, what does the output of the following section of code look like?

   ```
   GreetingRunnable r1= new GreetingRunnable("Hello, world!");
   GreetingRunnable r2=new GreetingRunnable("Goodbye, World!");
   r1.run();
   r2.run();
   ```
5. What does the **interrupt** method of the Thread class do? It does not make sense if another method is not used when the interrupt method is used, what is the other method that should be used when the interrupt method is used? Why?
6. What is the difference between the interrupted() method and the isInterrupted() method?
7. What does the start() method do when it is activated?
8. What happens when a thread is in its sleeping condition and is interrupted?
9. What is a racing condition?
10. In this chapter, we used a deposit thread and a withdraw thread to illustrate a racing condition. Give another example that illustrate a racing condition.
11. How to resolve a racing condition?
12. How to lock an object in multiple threading?
13. What does "locking an object" mean?
14. What is deadlock in multiple threading?
15. How do we avoid deadlocks in multiple threading?
16. Explain how each of the methods you mentioned in the previous problem works.
17. What are the two ways in Java used to lock an object?

Programming Exercises

1. Design an aquarium with several different colored fish moving in it. Your aquarium should be a window with water color background. Your moving fish should be implemented by using different threads. Your fish should move in the aquarium until the user closes the window. The fish shape can be defined by using the Polygon class and the fish color can be added by using the fillPolygon method from the Graphics class.
   The constructor of the Polygon class takes three parameters: int [] x, int [] y, and int n. The first array parameter should contain all the x coordinates of the corner points of the polygon, the second array parameter should contain all the corresponding y coordinates in order of the corner points, and the third parameter is the number of corner points. The fillPolygon method has a Polygon type parameter.
2. Design a multithread program that simulate the dining philosophers problem invented by E. W. Dijkstra. Imagine that five philosophers who spend their lives just thinking and easting. In the middle of the dining room is a circular table with five chairs and each philosopher sits on a chair. The table has a big plate of spaghetti. However, there are only five chopsticks available, and there is one chopstick between each pair of philosopher . Each philosopher thinks. When he gets hungry, he needs to

pick up the two chopsticks that are closest to him. If a philosopher can pick up both chopsticks, he eats for a while. After a philosopher finishes eating, he puts down the chopsticks and starts to think.

Chapter Six
Networking

Computer networking is the practice of linking two or more computer devices together for the purpose of sharing data and working on complicated problems by sharing resources. Networks are built with a mix of computer hardware and software.

In this chapter, we study several classes in the Java library that can be used for networking.

6.1 The URL and URLConnection Classes

In computer networking, machines use protocols to communicate. A protocol is a set of rules for two computers to communicate with each other. A client sends requests using commands provided by the protocol in the order specified in the protocol and the server responds similarly. URL is the acronym of uniform resource locator which has four parts: protocol name, the host address or domain, the port number and the path to the resource file. http, ftp and mailto are some of the protocol names. A domain name is a unique name that identifies an internet resource such as a personal computer and a server computer or a website. Domain names are formed by the rules of the Domain Name System. In general, a domain name represents an Internet Protocol resource, such as a personal computer used to access the Internet, a server computer hosting a web site or any other service communicated via Internet. A port number is an integer between 0 and 2^{16} which is used to distinguish the programs running on a machine.

The most basic class for networking in the Java library is the URL class. One of the URL constructors has a String parameter representing a url. One of its important methods is the openStream() method that opens a connection to this URL and returns an InputStream for reading from that connection.

The following is an example that allows the user to pass a URL through the command line, reads the file from the connection and prints it out.

```
import java.net.*;
import java.io.*;
public class TryURL
{
  public static void main(String[] args)
  {
   try
   {
      URL url=new URL(args[0]);
      Scanner input=new Scanner(url.openStream());
      while(input.hasNextLine())
         System.out.println(input.nextLine());
      input.close();
   }catch(Exception e){e.printStackTrace();}
  }
}
```

The URL class can be used to connect to a website and download a file, but the programmer cannot control the process of the connection. The URLConnection class makes a connection using a URL and adds methods for programmers to customize the connection and control the process.

Before we talk about the URLConnection class, let's first get familiar with the HTTP protocol. An HTTP client sends a request to a server in the following format: the first line contains method used, identifier of the resource and protocol version, and the following lines are various request headers which provide information about the capacity of the client. After the request headers comes the data to be sent to the server. Here is an example of a request:

Get /JPomfret/index.htm HTTP/1.0
User-Agent: java 1.4.2-03
Host: facstaff.bloomu.edu:80
Accept: text/html, image/gif, image/jpeg
Connection: keep-alive

An HTTP server responds to a request with a status line followed by various response headers. A status line contains HTTP version, status code and reason. Generally, status code has three categories: 2XX for success, 3XX for redirection and 4XX or 5XX for client error. The following is a sample response from the a server.

HTTP/1.1 200 OK
Server: Microsoft-11s/5.0
Microsoft Office Web Server: 5.0-Pub
Content Location: http://facstaff.bloomu.edu/jpomfret/index.htm
Date: Mon, 01 Mar 2004 19:46:41 GMT
Content Type: text/html
Accept Ranges: bytes.

The URL class has an instance method openConnection() that returns a URLConnection object. The constructor URLConnection(String url) can also be used to create a URLConnection object. The following methods of the URLConnection class can be used to get the server response.
InputStream getInputStream()—returns an InputStream for reading from this connection
String getHeaderFieldKey(int n)—Returns the key for the nth header field.
String getHeaderField(int n)—Returns the value of the nth header field.
The following is an example that allows the user to pass a URL through the command line, gets the server response and reads the file from the connection.

```
import java.net.*;
import java.io.*;
public class GetResponse
{
  public static void main(String[] args) throws Exception
  {
    URL url=new URL(args[0]);
    URLConnection c=url.openConnection();
    System.out.println(“Status Line: ”);
    System.out.println(‘lt’+c.getHeaderfield(0); //The status line does not
                                                  //have a key
    System.out.println(“Response Headers: ”);
    String value=null;
    int n=1;
   while(true)
   {
     value=c.getHeaderField(n);
     if(value==null)
       break;
     System.out.println(‘\t’+c.getHeaderFieldKey(n++)+ “: ”+value);
   }
     Scanner input=new Scanner(c.getInputStream());
     while(input.hasNextLine())
     System.out.println(input.nextLine());
     input.close();
  }
 }
```

6.2 Clients and Servers Using Sockets

The URL and URLConnection classes hide the details of the protocols so programmers can easily write programs to connect to a web server. The Java classes Socket and ServerSocket allow programmers to develop their own protocols for communications between client and server.

Here are one of the constructors and some methods of the Socket class.
Socket(String host, int port)—Creates a Socket object that is connected to the program running at the given port and the supplied host.
InetAddress getInetAddress()—Returns the address to which the socket is connected.
InputStream getInputStream()—Returns an input stream for this socket.
OutputStream getOutputStream()—Returns an output stream for this socket.

Since the socket is connected to a server, the input stream obtained by using the getInputStream() method can be used to read data from the server and the output stream obtained by using the getOutputStream() method can be used to send data to the server. Similarly, the ServerSocket class has the following constructor and methods.
ServerSocket(int port)—Creates a ServerSocket object bound to the specified port.
Socket accept()—Listens and waits for a Socket to connect. Once a Socket tries to connect, it returns a Socket object representing the client Socket.

After a Socket object is returned, the getInputStream() and getOutputStream() methods can be used to read data from and send data to the client.

Now, let’s design a fortune teller server program that may tell a client his/her fortune upon his/her request. If a client sends the word “today”, the server program would send a today fortune to the client. If a client sends the word “future”, the server program would send a future fortune to the client.

```
public class Fortune
{
  private ArrayList<String> todays; //A list storing today’s fortunes
   private ArrayList<String> futures;//A list storing future fortunes

   public Fortune()
   {
```

```
  todays=new ArrayList<String();
   futures=new ArrayList<String>();
   todays.add(“A friend is near”);
   todays.add(“Expect a call”);
   todays.add(“Someone misses you”);
    futures.add(“Wealth awaits—if you desire it”);
    futures.add(“Climb the hill of effort for high grades”);
    futures.add(“The door to success is open to you”);
  }

 /**Get a today’s fortune from today’s list
  *@return a random fortune from today’s list
  */
  public String getTodays(String  period)
  {
    Random gen=new Random();
    return todays.get(gen.nextInt(todays.size()));
  }

   /**Get a future fortune from the future list
  *@return a random fortune from the future list
  */

 public String getFutures(String  period)
  {
    Random gen=new Random();
    return todays.get(gen.nextInt(futures.size()));
  }
}

import java.net.*;
import java.io.*;

public class FortuneServer
{
 public static void main(String[] args) throws Exception
 {
  String s=null;
   Fortune f=new Fortune();
  ServerSocket server=new ServerSocket(5678);
```

```
   Socket client=server.accept();
    Scanner input=new Scanner(client.getInputStream());
   PrintWriter output=new PrintWriter(client.getOutputStream(), true);
   while(input.hasNextLine())
   {
     s=input.nextLine();
     if(s.ignoreCaseEquals("today"))
        output.println(f.getTodays());
     else if(s.ignoreCaseEquals("future"))
        output.println(f.getFutures());
   }
    input.close();
   output.close();
   client.close();
  }
}
```

Two issuess need to be pointed out. The first one is about the port number used. System servers use port numbers below 1024. Web servers use port number 80, SMTP for sending email uses port number 25 and POP3 for receiving email uses port number 110. To avoid stepping on somebody's toes, it is safer for a server socket to use port numbers larger than 1024. The second one is about the use of the constructor of PrintWriter. The PrintWriter class has several constructors. The one that takes an OutputStream as only one parameter waits for the buffer to be filled up and then flushes the data out. Because the output here is short, we need it to be flushed out without waiting for the buffer to get full. Thus, we need the constructor which has a boolean parameter as its second one and set its value to be true.

The corresponding client program is as following. Again, it takes the host address of the server through the command line. The host address could be an IP address or a regular url.

```
import java.net.*;
import java.io.*;

public class FortuneClient
{
 public static void main(String[] arge)
 {
  String s=null;
  Socket server=new Socket(args[0],  5678);
  System.println(“Connected to Fortuen server host: “
                                                  +server.getInetAddress());
  Scanner fromKeyboard=new Scanner(System.in);
  Scanner fromServer=new Scanner(server.getInputStream());
  PrintWriter toServer=new PrinteWriter(server.getOutputStream(), true);

 while(fromKeyboard.hasNextLine())
 {
  System.out.println(“Type a word to get your fortune, type the word ”
                    “‘today’ if you want a today’s fortune ”+
                    “ and type the word ‘future’ if you want a future fortune:”);
   toServer.println(fromKeyboard.nextLine());
  System.out.println(“Your fortune is ”+ fromServer.nextLine());
 }
  fromKeyboard.close();
  fromServer.close();
  toServer.close();
  server.close();
}
```

6.3 Threaded Server

Our fortune server in last section has very unusual behavior for a server in that it serves only one client because the accept() method there is executed once. To make it more useful, we need to put the server code in a loop and make each client a thread. In this way, our server can respond to as many clients as they connect and respond to their request concurrently.

```
import java.net.*;
import java.io.*;

public class FortuneServer
{
  public static void main(String[] args) throws Exception
  {
   ServerSocket server=new ServerSocket(5678);
   while(true)
   {
    Socket client=server.accept();
    new ClientThread(client);
    }
  }
}


public class ClientThread extends Thread
{
 private Socket client;
 private Scanner fromClient;
 private PrintWriter toClient;
 public ClientThread(Socket c) throws Exception
 {
   client=c;
   fromClient=new Scanner(client.getInputStream());
   toClient=new PrintWriter(client.getOutputStream(), true);
   start();
 }
 public void run()
 {
  try
  {
   String s=null;
    Fortune f=new Fortune();
   while(fromClient.hasNextLine())
   {
     s=fromClient.nextLine();
     if(s.ignoreCaseEquals("today"))
        toClient.println(f.getTodays());
    else if(s.ignoreCaseEquals("future"))
        otoClientt.println(f.getFutures());
   }
    fromClient.close();
   toClient.close();
   client.close();
  }catch(Exception e){e.printStackTrace();}
 }
}
```

Review Questions

1. How many parts does a url have" ? What are they?

2. How many and what type of parameters does the ServerSocket constructor have? What is each of them for
3. Which class is the accept() method from? What does it do?
4. How many and what types of parameters does the Socket constructor have? What does the Socket constructor do?
5. After a server is connected to a client, how to read data from the client?
6. After a server is connected to a client, how to send data to the client?
7. After a client is connected to a server, how to read data from the server?
8. After a client is connected to a server, how to send data to theserver?

Programming Projects

1. WeChat has been a very popular program. This program allows both one-to-one chat and group chat. Design a server for this program. Design your client class and test your server. Right after a client connects to the server, the server should send a list of the clients including the ones who are "online" to the client. Then, your client may pick any one or a group from the list to chat with by sending the name followed by a slash and the message to the server. The server then inspects the name and message and sends the message to the correct client with the name of the sending client in front of the message.

   a. This clearly has to be a threaded server for many clients to chat through it.
   b. To remember the clients who have connected, you may need **a map** that associates each client (represented by a Socket object) connected to the server to a given name (string). The name is what you use to reference the client. You may ask the client to supply his/her name and change the name you give to the name supplied by the client.
   c. You may need to have **a thread** for reading from and **another thread** for writing to each client.
   d. Include professional documentation and appropriate indentation.
   e. In front of your server program, use comments to describe **the protocol** that governs the communications between your server and clients.
   f. Test your server and produce a sample of chatting list.
   g. For this project, using console to chat is acceptable although graphics window is preferred.
   h. Right now, you may use a list to store the users and their messages. You may use a database after we study JDBC in next chapter.

2. Use Socket and ServerSocket to design a program that allows two players to play the regular tic-tac-toe game. The server has the tic-tac-toe board, modifies it and sends it to the other player when a player makes a move.

3. Use Socket and ServerSocket to design a program that allows two players to play the regular chess game. Your server contains the chess board with the

pieces in position, moves the piece and sends the board to the other player when a player drags a piece from one position to another. If a player makes an illegal move, your program should pop up a warning window to alert the player to make another one without moving the piece.

Chapter Seven
Java Database Connection (JDBC)

For small applications, we can use files to store data, but as the amount of data that we need to save gets larger, services of a database system become invaluable. A database is a collection of data that is organized in such a way that adding, managing and retrieving information are made easy. A database management system is a piece of software that allows programmers to model the information needed while it handles the details of the inserting, removing and retrieving data from individual files in response to the request of the users. The JDBC programming interface hides the details of different databases so programmers can work with many different databases.

7.1 Database Tables and SQL Queries

In a relational database, we keep our data in tables. A row in a table is called a record and a column is called a field. A primary key of a table is a field or a group of fields that can be used to identify each record uniquely. The relational database model uses multiple tables opposed to historical databases what were one giant flat file table. The allowance of using multiple tables reduces the data redundancy of one giant tale. In a relational database, multiple tables are linked by matched fields so that two tables can be queried simultaneously by the users.

Suppose that we have a company and we want to store our orders in a database. An order has one customer who can order several items. A salesperson may take several orders from the same customer, but each order is taken by exactly one salesperson.

We first need a table (Customer) to describe all customers that the company has.

CustomerID	CustomerName	Address	BalanceDue

Since different customers may have the same name, we assign each customer a unique ID in this table so that each customer can be identified uniquely. Thus, the CustomerID field can be used as the primary key of this table. We also need a Salesperson table to describe every salesperson and an ItemTable to describe every item available in the company.

SalespersonID	SalespersonName	Address	PhoneNumber

ItemID	Description	Quantity

We clearly need an OrdersTable to describe the orders. The problem is that an order may have multiple items so we use one table (OrdersTable) to record the general information of all orders and use a second one (OrderItem) to list the items in each order.

OrderID	CustomerID	SalespersonID	OrderDate

OrderID	ItemID	Quantity	UnitPrice

In the OrderItem table, no single field can be used to identify every record uniquely so we need a compound key that consists of OrderID and ItemID.

SQL, the acronym of structured query language, is a standard language used to get information from or make changes to a database. SQL is case insensitive, but we will make every keyword uppercase for distinguishing it from other words. SQL has two categories of statements generally: update and query.
A CREATE statement is used to create a table; an INSERT statement is used to insert a record into a table; a DELETE statement is used to remove some records from a table; an UPDATE statement is used to change the values of some fields of some records of a table; a SELECT statement is used to query out some information from a database. Here are the general syntax of these statements.

CREATE TABLE table_name(field_name1 type, ……)

INSERT INTO table_name VALUES(value1, ……)

DELETE FROM table_name WHERE logical_statements

UPDATE table_name SET field_name1=value1, … WHERE logical_statements

SELECT field_name1, …… FROM table_name WHERE logical_statements

Some other statements include the ALTER statement:
ALTER TABLE tableName ADD colName colType;
The types depend on the database management system. The following are some common types.

CHAR(N)—a fixed size string of length N. The range of N is 0 to 255. If N is omitted, the length is 1.

VARCHAR(N)—a variable size string of length up to N. The range of N is 0 to 255.

BOOL, BOOLEAN --A value of zero is considered false. Nonzero values are considered true.

INT, INTEGER—A normal-size integer. Its range is -2147483648 to 214748364.
FLOAT(M, D)—A small floating-point number. M is the total number of digits and D is the number of digits following the decimal point. If M and D are omitted, values are stored to the limits of the hardware.

DOUBLE(M, D)—A normal-size floating point number.

DATE—A date in the format of YYYY-MM-DD.

A primary key of a table can be designated in a CREATE statement by putting the keywords PRIMARY KEY at the end of the parentheses followed by the list of fields for the key separated by commas if needed within another pair of parentheses.

The following are some examples of simple SQL statements that work with the tables we described earlier.

```
CREATE TABLE Customer(CustomerID CHAR(4), CustomerName
                VARCHAR(25), Address, VARCHAR(25),
                BalanceDue DOUBLE, PRIMARY KEY(CustomerID));
INSERT INTO Customer VALUES(1234, ‘Youmin Lu’, ‘100 Lucky St.’,
                                        2000.00)
```

If a string value does not have spaces, quotes (single or double) are optional, but you have to place a pair of quotes to ensure that it is a single value if it contains some spaces. If the statement is not in quotes, either single or double quotes are appropriate. If it is already in double quotes, single quotes are needed.

```
DELETE FROM Customer WHERE CustomerID=’1234’;
```

You may use the relational operators BETWEEN …AND…, =, !=, >, <, >= and <= to create logical statements and the keywords AND, NOT and OR to combine several simple logical statements into a compound one.

```
UPDATE Customer SET BalanceDue=3000.00 WHERE CustomerID=’1234’;
SELECT * FROM Customer;
SELECT CustomerName, Address FROM Customer;
```

If the WHERE clause is omitted in a select statement, it means that there is no condition needed so that the fields listed are queried out from all records of the table. If all fields are needed, an asterisk * can be used without listing any field name.

SELECT CustomerName, Address from Customer WHERE CustomerID=’1234’; SELECT CustomerName, Address FROM Customer, ORDERS

```
        WHERE Customer.CustomerID=Orders.CustomerID AND
                                  OrderDate='2004-03-31';
```

7.2 Connecting to a Database and Executing SQL Statements

To connect a Java program to a database in JDBC, we need **add the appropriate connector to the project library** and use the static method getConnection(String url, String user, String pw) from the DriverManager class to obtain a Connection object. In the earlier version of JDBC, we need to load the appropriate driver first by creating an object of the driver. For using Microsoft Access, you may either use the statement

```
      new JdbcOdbcDriver();
```

or

```
   Class.forName("JdbcOdbcDriver");
```

The statements

```
   String url="jdbc:odbc:YLU";
   String user="YLU";
    String pw="YLU";
   Connection c=DriverManager.getConnection(url, user, pw);
```

would get a Connection object connected to the Microsoft Access database named as YLU.

Similarly, the statement

```
     new com.mysql.jdbc.Driver(); //it is not needed in the newer version
```

or

```
      Class.forName("com.mysql.jdbc.Driver");
```

can be used to load the driver for using a MySql database.

The statements

```
  String url=
    "jdbc:mysql://classdb.mads.bloomu.edu:3306/ylu";
 String user="ylu";
  String pw="ylu";
 Connection c=DriverManager.getConnection(url, user, pw);
```

Would get a Connection object connected to the database with name ylu on the port 3306 in the machine at classdb.mads.bloomu.edu. If you are using NetBeans IDE, you may also need to download the connector from http://dev.mysql.com/downloads/connector/J and add the MySql driver to the library (right click the project name, select properties, select library, select add library, select MySql driver in the list and click the add button).

Once a Connection object is obtained, you may use its instance method createStatement() to get a Statement object:

```
   Statement stmt=c.createStatement()
```

and use the following methods from the Statement class to execute SQL statements. There are two Statement classes in the Java library; we need the one from the java.sql package.

int executeUpdate(String sql)—Executes the given SQL statement, which may be a CREATE, INSERT, UPDATE, OR DELETE statement.

ResultSet executeQuery(String sql)—Executes the given SQL statement and returns a ResultSet object that contains the information queried out.

Here are some example statements that execute some SQL UPDATE statements.

```
String sql=” CREATE TABLE Customer(CustomerID CHAR(4), “+
                 “CustomerName VARCHAR(25), ”+
                 “ Address, VARCHAR(25), BalanceDue DOUBLE, “+
                 “PRIMARY KEY(CustomerID))”;
stmt.executeUpdate(sql);
sql=” INSERT INTO Customer VALUES(1234, ‘Youmin Lu’, ”+
      “ ‘100 Lucky St.’, 2000.00)”;
stmt.executeUpdate(sql);
sql= “UPDATE Customer SET BalanceDue=3000.00 “+
                  “WHERE CustomerID=’1234’”;
stmt.executeUpdate(sql);
stmt.executeUpdate( “DELETE FROM Customer ”+
                            “WHERE CustomerID=’1234’”);
```

To show more examples, let’s create our orders table and add a few records into it.

```
sql= “CREATE TABLE Orders(OrderID CHAR(4), ”+
       “CustomerID CHAR(4), SalesPersonID CHAR(4), ”+
       “OrderDate DATE, PRIMARY KEY(OrderID))”;
stmt.executeUpdate(sql);
stmt.executeUpdate(“INSERT INTO Orders VALUES”+
                            “(‘1111’, ‘1234’, ‘1212’, ‘2014-01-01’)”);
stmt.executeUpdate(“INSERT INTO Orders VALUES”+
                            “(‘2222’, ‘2345’, ‘1212’, ‘2014-03-01’)”);
stmt.executeUpdate(“INSERT INTO Orders VALUES”+
                            “(‘3333’, ‘1234’, ‘2323’, ‘2014-08-12’)”);
stmt.executeUpdate(“INSERT INTO Orders VALUES”+
                           “(‘4444’, ‘2345’, ‘2323’, ‘2014-12-31’)”);
```

The method executeQuery(String sql) returns a ResultSet object representing the set of the records queried out. The ResultSet class is from the java.sql package. The set of records can be viewed as a table again and an imaginary cursor is pointing to a mark before the beginning of the set. That means, the cursor does not point to the first record of the set and you need to move the cursor once for getting the first record from the set. The first important method of the ResultSet class is the next method:

boolean next()—Returns true if there are more records after the cursor in this set and advances the cursor to the next record, otherwise, false is returned.

There are two methods for retrieving data in the ResultSet class for each data type:

XXX getXXX(String field_name)—Returns the value of the field with the given name in the record pointed to by the cursor in this ResultSet.

XXX getXXX(int field_index)-- Returns the value of the field with the given index in the record pointed to by the cursor in this ResultSet. The index starts from 1.

Now, let’s try to execute some QUERY statements.

```
sql= “SELECT CustomerID, OrderDate FROM Orders ”+
        “WHERE OrderDate BETWEEN ‘2014-02-20’AND ‘2014-08-20’”;
ResultSet rs=stmt.cuteQuery(sql);
```

To display all the records from the set, one may use a while loop to move the cursor.

```
while(rs.next())
   System.out.println(rs.getString(“CustomerID”)+ “\t”+rs.getDate(2));
```

In this statement, we used the field name in one get method and the index in another for the purpose of testing.

7.3. Prepared Statements

A PreparedStatement is an object that contains an SQL statement template with some values missing. If you want to execute a statement object many times, using a PreparedStatement may reduce execution time. A PreparedStatement is given an SQL statement when it is created so this SQL statement is sent to the DBMS right away, where it is compiled. As a result, the PreparedStatement contains a SQL statement that has been compiled so the DBMS can just run the PreparedStatement SQL statement without having to compile it first when the preparedStatement is executed.

A PreparedStatement object can be used for a regular SQL statement, but it makes more sense to use a PreparedStatement for SQL statement with parameters for missing values. Before you supply a SQL statement with missing values, you use one question mark to represent each missing value in your SQL statement.

Once you have an SQL statement with or without parameters, you can use the following instance method from the Connection class to create your preparedStatement:
PreparedStatement prepareStatement(String sql).

If your PreparedStatement contains an SQL statement with parameters, you need to use the following instance method from the PreparedStatement class to supply values in place of the question mark palceholder before you may execute your PreparedStatement:
void setXXX(int index, XXX value)—Sets the value to the parameter at the given index, where index starts from 1, not 0 and XXX should be the type of the value.

After all parameters of a PreparedStatement have been set with values, it retains the values until they are reset to other values, or the method clearParameter is called. Once all parameters of a PreparedStatement are set, it can be executed using one of the following instance methods from the PreparedStatement class:
boolean execute()—Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement.
ResultSet executeQuery()—Executes the SQL query in this PreparedStatement object and returns a ResutlSet obejct containing the data queried out.
int executeUpdate()—Executes the SQL statement in this PreparedStatement object, which must be an SQL data manipulation statement such as INSERT, UPDATE or DELETE.

For example, you may have

```
String sql="INSERT INTO Orders VALUES(?, ?, ?, ?)";
PreparedStatement ps=c.prepareStatement(sql);
ps.setString(1, '9999');
ps.setString(2, '7889');
ps.setString(3, '9887');
ps.setDate(4, Date.valueOf("2015-05-31"));
ps.execute();
```

Make sure that you need to use the valueOf(String date) method from the java.sql package, not the one from the java.util package.

7.4 Meta Data

We may need to know some general information about the database such as table names in the database and column names of each table. These information are called metadata. To get metadata from a database, we first need to obtain a DatabaseMetaData object by using the method getMetaData() from the Connection class:

```
DatabaseMetaData dbmdata=c.getMetaData();
```

The ResultSet class also has a method getMetaData() which returns an object of ResultSetMetaData that contains the information about the ResultSet. Now, we can use the method getTypeInfo() to get the information about the types supported by the database we are using.

```
ResultSet rs=dbmdata.getTypeInfo();
ResultSetMetaData rsmtData=rs.getMetaData();
int cols=rsmtData.getColumnCount();
while(rs.next())
{
  for(int i=1; i<=cols; i++)
     System.out.print("\t"+rs.getObject(i));
  System.out.println();
}
```

Two other important methods from the DatabaseMetaData class are the getTables and getColumns mehtods:
ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)—Searches the database for tables that match the given catalog, schemaPattern, tableNamePattern and the list of types supplied

through the parameters and return a ResultSet containing the description of the tables. Each table description ResultSet has 10 columns: TABLES_CAT, TABLE_SCEM, TABLE_NAME, TABLE_TYPE, REMARKS, TYPE_CAT, TYPE_SCHEM, TYPE_NAME, SELF_REFERENCING_COL_NAME and REF_GENERATION. The first parameter catalog should be a catalog name that matches the catalog name as it is stored in the database; null can be used to indicate that the catalog name should not be used to narrow the search. The second parameter schemaPattern needs to match the schema name as it is stored in the database; null can be used to indicate that the schema name should not be used to narrow the search. The third parameter tableNamePattern needs to match some table names as they are stored in the database; a percentage sign % can be used to represent many characters and a underscore symbol _ can be used to represent a single character. The last parameter is an array which needs to be from the list of the table types returned from getTableTypes(); null returns all types.
ResultSet getColumns(String catalog, String schemaPattern, String tableNamePaqttern, String columnNamePattern)—searches through the database for the columns of the tables that match the given information and returns a ResultSet containing the description. The first two parameters are the same as those in the getTables method. The third one is usually a table name and the fourth one can be null for getting all the columns of the given table in the third parameter. The ResultSet returned contains 24 columns and the most important one is COLUMN_NAME.

Here are some examples of using these methods.

```
String[] arr={"TABLE"};
rs=dbmdata.getTables(null, null, "%", arr);
while(rs.next())
   System.out.println(rs.getString("TABLE_NAME");
rs=dbmData.getColumns(null, null, "Customer", "%");
while(rs.next())
   System.out.println(rs.getString("COLUMN_NAME");
```

7.5 Aggregate SQL Functions and Transactions

An aggregate SQL function is used in an SQL statement for computing a numerical data against a column/field.
MIN(fieldName)—Returns the minimum value of the given column.
MAX(filedName)—Returns the maximum value of the given column.
SUM(fieldName)—Returns the total value of the given column.

AVG(fieldName)—Returns the average value of the given column.
COUNT(fieldName)—Returns the total number of values in the given column.
COUNT(*)—Returns the total number of rows in a table.
The following is an example of using these functions.

```
String sql=”SELECT COUNT(balance), MAX(balance), AVG(balance) ”+
              “FROM YluStudents WHERE balance>=1000”;
ResultSet rs=sm.executeQuery(sql);
rs.next();
System.out.println(“Number of balances>=1000: ”+rs.getObject(1)+
                            “\nThe average: ”+rs.getObject(3)+
                           “\nThe maximum: ”+rs.getObject(2));
```

Recall that the imaginary cursor of a ResultSet is not at the first record. To get the data from the first record, you need to call its next method once for moving the cursor to the first record.
Now, let’s consider transactions. There are times when you don’t want one statement to take effect unless another one completes. For example, when you want to update the balance of an account after a client has made a deposit, you also want to update the table that records the transactions. The way to make sure that either both actions occur or neither actions occur is to use a transaction.
A transaction is a set of one or more SQL statements that is executed as a unit, so either all of the statements in the set are executed and applied to the database or none of them is executed and applied to the database. In JDBC, a connection is in auto-commit mode when it is created: each SQL statement is treated as a transaction and is automatically committed (makes change to the database) right after it is executed. To allow two or more statements to be grouped into a transaction, you need to
1). Disable the auto-commit mode by using the following instance method from the Connection class:

```
        void setAutoCommit(false);
```

2). Call the following instance method of the Connection class when you are ready to make all changes from a transaction:

```
      void commit().
```

After the auto-commit mode is disabled, no SQL statements are committed until you call the commit method explicitly. All statements executed after the previous call to the commit method are included in the current transaction and committed together as an entire unit.
The following instance method of the Connection class can be combined with a try-finally block to guarantee that none of the SQL statements in a transaction will be executed when any one of them has an exception:
void roolback().

In this case, you need to place the commit method call at the end of the try-block and the rollback method call in the finally block.

Review Questions

1. What is a database?
2. What is the advantage of a relational database?
3. What is a database management system?
4. What is a row called in a database table? What is a column called in a database table?
5. What is a primary key of a database table? What is it for?
6. What is SQL?
7. What is the difference between the SQL types CHAR(int n) and VARCHAR(int n)?
8. There are five types of SQL statements. What is each of them for?
9. What is the general syntax of a SQL SELECT statement?
10. What are the two steps needed for connecting a java program to a database?
11. After a program is connected to a database, it needs to use methods from the Statement class to access or modify the database. How should you get a Statement object?
12. Describe carefully the two methods that we have learned from the Statement class.
13. The ResultSet class has a method next(). What does this method do when it is activated?
14. Describe the two types of get methods from the ResultSet class carefully(return type, name, parameters and what they do).
15. How should you convert an incomplete SQL statement (an SQL statement that contains question marks) into a PreparedStatement?
16. How to complete (change a question mark to a value) an SQL statement stored in a Preparedstatement?
17. Once an SQL statement stored in a preparedstatement is complete, it can be executed. Describe a method (including name, return type, parameters, the class in which it is defined, and what it does) used to execute PreparedStatement.
18. The JDBC default is to commit the change to the database as soon as an SQL statement is executed. Why do we want to change from this default to require that we explicitly commit changes ?
19. How do we change from the JDBC default to require that we explicitly commit changes?
20. What method (including name, return type, parameters, the class in which it is defined, and what it does) should you use to commit the changes to the database after the JDBC default has been changed?
21. If you have executed some SQL update statements and decided not to commit them, you may use the Connection class method rollback(). To what point does it roll back when this method is used?

Programming Projects

1. Use Java GUI and JDBC to design a program that supports electronic banking. First, you need to have a database that may contain the information of the customers. This database needs to be well designed with appropriate table structures that contain all customer information including customers’ id, User Name, Password, name, address, the balance of each of the three accounts (Saving, Checking, and Money Market) of each customer if s/he has any, and the transaction information on each account made by each customer. Your GUI program will allow the customers to open a new account, deposit to or withdraw from an account, transfer from one account to another and list all transactions made to an account within a certain time period supplied by the customer. When your program is executed, it should ask the customer if s/he wants to login or open a new account. If a customer wants to open a new account, you should ask the customer to enter his/her name address, a user name, a pass word, what kind(s) of account(s) to open and how much to deposit into each new account. Of course, you should store all information supplied by the customer into your database. If a customer wants to deposit some money, your program should show a list that includes all the accounts the customer has so s/he can select an account depositing into. If a customer wants to withdraw, your program should show the list of accounts the customer has and the corresponding balance in each account so that your customer may select an account to withdraw from. If a customer wants to look at the transactions within a period, you should ask the customer to select an account, enter the beginning date and ending date of the period. If a customer wants to transfer money from one account to another, your program should show the list of the accounts with the balances that the customer has and allow the customer to select the one s/he wants to transfer from, then show the list again for the customer to select the one s/he wants to transfer to.
2. Programming Project number 1 in chapter 7 stores clients’ data in a list so the data would be lost when the server is off. Add database to the project so the data can be stored permanently.

Chapter Eight
Servlets

A **servlet** is a Java class that is used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Serlets are commonly used to extend the applications hosted by web servers and define HTTP-specific servlet classes.

The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods. When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.

The HTML language can be used to structure a webpage static content and CSS can be used to style the page. In this case, a user types a URL for the file from a web browser, the browser contacts the web server and requests the file, the server finds the file and returns it to the browser and finally the browser displays the file to the user. HTML files store static information which may be updated periodically by the programmers. But at any given time, every request for the same document returns exactly the same result. Much information such as stock quotes, weather and bank account are not static in nature and an ideal webpage should be able to run certain programs to process user requests from web browsers and produce customized and on time response. This kind of webpages is called dynamic pages and Java servlets are created for this purpose.

8.1 HTML Basics

HTML, the acronym of HyperText Markup Language, is the standard markup language used to structure the content of webpages. The latest version of HTML is HTML5. A browser is a software application for retrieving, presenting and traversing information resources from the World Wide Web. All browsers understand the HTML language and can interpret webpages in HTML and display their content. HTML consists of tags and most tags go by pairs in which a start tag has a command enclosed in angle brackets and the corresponding end tag is only different by a forward slash in front of the tag command within angle bracket. Between a start tag and its corresponding end tag is the content marked by the tags. Basically, the tags tell the browsers how to display the contents enclosed by the tags. Some HTML tags are empty tags without contents and so are unpaired. An empty tag has the command followed by a forward slash within angle bracket. Because an empty tag does not have contents, its purpose is to mark up a point of the page such as changing line. Many HTML tags in the earlier versions are not supported by HTML5. If one wants to set the effects/style of these tags, he/she needs to use CSS, the acronym of Cascade Style Sheet, which will be covered briefly in

next section. Since this is a Java course, we only cover the basic HTML needed in this course.
HTML is case insensitive; thus, you may type a command in upper case, lower case or a mixture of them. To distinguish the content from the commands, we will always type the commands in upper case.

An HTML5 document must start with the document declaration:
<!DOCTYPE HTML>.
Following the document declaration is the major part of the document which is enclosed in the tags: <HTML>……</HTML>. The major part of a document is divided into two parts: the HEAD part and the BODY part. The HEAD part is enclosed in the tages: <HEAD>…</HEAD> and similarly the BODY part is enclosed in the tags:
<BODY>…</BODY>. So a complete HTML document looks like

```
<!DOCTYPE HTML>.
<HTML>
<HEAD>  head content </HEAD>
<BODY> This is my first page</BODY>
</HTML>
```

An HTML start tag may contain attributes following its command and each attribute has a value assigned by using the assignment symbol =. For example, the BODY tag may have the attribute BGCOLOR that is used to give a color to the background of the page.
There are two types of values for the BGCOLOR attribute: one is the actual color word such as red, yellow, blue, purple, and so on, and the other is color code. Basically, every color word means what it means literally. The color code is given by a number sign # followed by six hex decimal digits. A hex decimal digit is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. So A, B, C, D, E and F are used to represent 10, 11, 12, 13, 14, and 15 respectively. The color code is a color plate that needs you to put certain amount of each of the three primary colors (red, green and blue) to make a new color. The first pair of digits is used to indicate how much red color needed; the second pair is used for green and the third pair for blue. Of course, 00 is the least and FF is the most. For example, #FFFFFF represents white color, #000000 represents black color, #FF0000 represents red color and #FFFF00 represents yellow color. You may use the following code to make the background of your page red:
<BODY BGCOLOR=RED>……</BODY>
or <BODY BGCOLOR=#FF0000>……</BODY>

The content of the HEAD section of a webpage displays little to human viewers and it is more important for the spiders or bots. Thus, it is essential to design the HEAD part carefully if the page needs to be visible to the search engines. The single most important thing for Search Engine is to write a meaningful and realistic title tag
<TITLE>……</TITLE>. The content of the TITLE tag does not appear on the page itself, but does show in the bar across the top of the browser. Like the other languages, HTML allows comments which are only for the programmers and are

ignored by the browsers. HTML comments are enclosed in <!--coments -->. For example, <!--This is a comment which is ignored by the browsers -->

The content of the body part is the information displayed by the browser. The tag <BR/> is an empty tag which is used to break a line or to change line. The browsers don't recognize multiple spaces and changing lines in default. For example, both the code

```
<BODY>
                 This is the first line of my web page.
                 This is the second line of my web page.

</BODY>
```

and

```
<BODY>
       This is the first line of my web page.       This is
       the second line of my web page.
 </BODY>
```

provide the same display:
This is the first line of my web page. This is the second line of my web page.
The correct code should be

```
<BODY>
     This is the first line of my web page.<BR/>
     This is the second line of my web page.
</BODY>
```

Both the tag <B> and <STRONG> make the same effects for human eyes, but the tag <STRONG> makes its content more important to the bots. For example,
<B>This part is in bold face</B>
<STRONG>This part is also in bold face, it also attracts more attention of the bots</STRONG>

Heading tags are used to enclose titles and subtitles. There are six levels of heading tags which starts from <H1>……</H1> with its content displayed large to <H6>……</H6> for the smallest. We usually place titles in the <H1> tags and subtitles in the other five heading tags.

The paragraph tags <P>……</P> are used to enclose a paragraph as what it literally means. There are two elements that are similar to the <P> element: <DIV> and <SPAN>. The <DIV> is a block tag which changes line before and after its content. The difference of the <DIV> tag from the <P> tag is that the <P> tag leaves an empty line before and after its content, and the <DIV> tag does not. The <SPAN> tag is an inline tag so basically it does not do anything unless CSS code is used to style its content. Thus, both the <DIV> and <SPAN> tags are dummy ones for using CSS code to style their content.

There are three kinds of lists in HTML: unordered list, ordered list and definition list. An ordered list is numbered from one and an unordered list is bulleted. A definition list has a definition following each term to be defined. The tags for unordered list are <UL>…..</UL>, for ordered list <OL>……</OL> and for definition list <DL>……</DL>. In the ordered and unordered lists, each list item is enclosed in the tags <LI>……</LI>. In a definition list, each list item has a term which should be enclosed by the tags <DT>……</DT> and its definition which should be enclosed in the tags <DD>……</DD>. For example,

```
<STRONG>Computer Science Classes in the Fall of
2020</STRONG><BR/>
<OL>
   <LI>Graphical User Interface in Java</LI>
   <LI>Data Structures using C++</LI>
   <LI>Internet Programming</LI>
</OL>

<STRONG>Computer Science Classes in the Fall of
2020</STRONG><BR/>
<UL>
   <LI>Graphical User Interface in Java</LI>
   <LI>Data Structures using C++</LI>
   <LI>Internet Programming</LI>
</UL>

<STRONG>Some Interesting Words</STRONG><BR/>
<DL>
     <DT>Auspicious</DT>
          <DD>Giving or being a sign for future
success</DD>
  <DT>Altruistic</DT>
        <DD>Showing unselfishly concern for or devotion to
the welfare of others</DD>
  <DT>Acquiesce </DT>
         <DD>To assent tactically; submit or comply silently
or without protest</DD>
</DL>
```

The TABLE tags are used to display information in rows and columns. Inside the TABLE tags <TABLE>……</TABLE> are <TR>……</TR> or table row tags which in turn contain <TD>……</TD> or table dividing tags. Data are enclosed in the <TD>……</TD> tags. Here is an example for a simple table without border line.

```
<TABLE>
  <TR>
      <TD>Row 1, Column 1</TD>
      <TD>Row 1, Column 2</TD>
  </TR>
  <TR>
      <TD>Row 2, Column 1</TD>
      <TD>Row 2, Column 2</TD>
  </TR>
</TABLE>
```

The TABLE tag allows an attribute BORDER which is used to specify the width of the border of the table. In default, all the rows have the same number of columns and all the columns must have the same number of rows in a table. If you want to modify the default, you may use the COLSPAN attribute in a <TD> tag to indicate how many columns this cell spans into. Similarly, you may use the ROWSPAN attribute in a <TD> tag to indicate how many rows this cell spans into. For example, the following code displays a table with one column in the first row and two columns in the second row.

```
<TABLE BORDER=”1”>
  <TR>
      <TD COLSPAN=”2”>Row 1, Column 1 & Column 2</TD>
  </TR>
  <TR>
      <TD>Row 2, Column 1</TD>
      <TD>Row 2, Column 2</TD>
  </TR>
</TABLE>
```

Row 1, Column 1 & Column 2	
Row 2, Column 1	Row 2, Column 2

Similarly, the following code displays a table with two rows in the first column and one row in the second column.

```
<TABLE BORDER=”1”>>
  <TR>
      <TD>Row 1, Column 1</TD>
      <TD ROWSPAN=”2”>Row 1 & Row 2, Column 2</TD>
  </TR>
  <TR>
      <TD>Row 2, Column 1</TD>
  </TR>
</TABLE>
```

Row 1 & Column 1	Row 1 & Row 2, Column 2
Row 2 & Column 1	

Any row or column can span at any position. Here is another example.

```
<TABLE BORDER="1">
   <TR>
      <TD rowspan="2">Row 1 & Row 2, Column 1</TD>
      <TD>Row 2, Column 2</TD>
      <TD colspan="2">Row 1, Column 3 & Column 4</TD>
   </TR>
   <TR>
     <TD rowspan="2">Row 2 & Row 3, Column 2</TD>
     <TD>Row 2, Column 3</TD>
     <TD>Row 2, Column 4</TD>
  </TR>
  <TR>
  <TD>Row 3, Column 1</TD>
  <TD>Row3, Column 3</TD>
  <TD>Row 3, Column 4</TD>
  </TR>
</TABLE>
```

Row 1 & Row 2, Column 1	Row 2, Column 2	Row 1, Column 3 & Column 4	
	Row 2 & Row 3, Column 2	Row 2, Column 3	Row 2, Column 4
Row 3, Column 1		Row3, Column 3	Row 3, Column 4

The empty tag <IMG/> or image tag can be used to insert an image into your page. Its attribute SRC is to be used to specify the file path of the image inserted; its ALT attribute may give some text which shows up when the picture does not show; the WIDTH and HEIGHT attributes which are optional can specify the dimensions of the picture. The following code inserts the picture in the file picture.jpg into the webpage.

```
<IMG SRC= "picture.jpg" ALT= "The picture should be here"
        WIDTH= "300" LENGTH= "200"/>
```

The tag <A>……</A> or anchor tag can be used to link your page to another page or your email depending on the protocol in the value of its HREF attribute. Its content is displayed in color and a viewer may click on the displayed content and move to another page. For example, the following code is to bring the viewers to the GOOGLE page when it is clicked.

```
<A HREF= http://www.google.com> Click to go to GOOGLE</A>
```

The following code would allow you to send me an email if you click on the displayed message.

```
<A HREF= mailto:ylu@bloomu.edu> Click to send me an
email</A>
```

An HTML form is a section of HTML code that contains normal content, markup, special elements called controls and labels on those controls. The general purpose of a form is for the viewers to modify its controls and submit it to a Web server or mail server to process the information.

As usual, a form is started by the tag <FORM> and ended by the tag </FORM>. The form tag has two attributes: ACTION and METHOD. The value of the ACTION attribute is the URL of the server to which the data of the form is going to be sent. The METHOD attribute may have one of the two values: get or post. A value get means that the data from the form is going to be sent to the server through the URL (all data is appended to the URL and sent to the server) while a value post means the data is going to be sent to the server through a package.

```
<FORM ACTION= "serverServlet.java" METHOD= "GET">
  Form elements are here
</FORM>
```

There are three basic control elements in a form: INPUT, SELECT and TEXTAREA. An INPUT element may be used to create a textbox, password box, checkbox, radio buttons, submit and reset buttons. An SELECT element can be used to create a scrolled list box from which the viewers can choose. A TEXTAREA is an area for the viewers to enter large amount of text.

An INPUT element may have the following attributes: type, name, value, checked and size. The value of the type attribute determines what this INPUT is and its value can be text, password, checkbox, radio, submit or reset. A value text for the type attribute makes the INPUT a textbox; a value password makes it a textbox with hidden value; checkbox or radio value makes an element indicated by its literal meaning and a group of them can be mutually exclusive if they share the same name; a value submit makes the INPUT a button and the data in the form is sent to the server specified by the value of the ACTION attribute of the FORM when it is clicked by the viewer; a value reset of the type attribute makes a button and all the values in the form elements are changed back to the default when it is clicked by the viewer. The value of the name attribute of an INPUT serves as its identifier and the server which accepts data from the form uses this value to distinguish this element from the others. The value of the value attribute of an input is what this element contains; at the moment the element is created, it contains the value (default value) entered by the programmer; and then it contains the value entered by the viewer later. The checked attribute is only used for a checkbox or radio and it does not have a value. The size attribute is used only for the text and password INPUT and it is used to specify how many characters the INPUT can hold.

```
<FORM ACTION= "/servlet/pizza" MEHTOD= "GET">
   <STRONG>Welcome to LU's pizza</STRONG><BR/>
   User Name:
     <INPUT TYPE= "text" NAME= "username" SIZE= "20"/>
   Password:
     <INPUT TYPE= “password” NAME= “password” SIZE= “20”/>
   Select the size of the pizza:
    <INPUT TYPE= “radio” NAME= “size” VALUE= “16” CHECKED/>
    16 Inches
    <INPUT TYPE= “radio” NAME= “size” VALUE= “12”/>
    12 Inches
    <INPUT TYPE= “radio” NAME= “size” VALUE= “6”/>
    6 Inches<BR/>
    Select the toppings:
      <INPUT TYPE=“checkbox” NAME=“toppings”
             VALUE=“mushroom” CHECKED/> Mushroom
      <INPUT TYPE=“checkbox” NAME=“toppings”
             VALUE=“pepperoni” CHECKED/>Peperoni
      <INPUT TYPE=“checkbox” NAME=“toppings”
             VALUE=“sausage” CHECKED/>Sausage<BR/>
  <INPUT TYPE= “submit” NAME= “Submit”/>
  <INPUT TYPE= “reset” NAME=”Reset”/>
</FORM>
```

A SELECT element/tag may have the following attributes: name, size and multiple. As in the INPUT element, the value of the name attribute is used by the server to identify this element. The value of the size attribute is to be used to indicate how many items in the list should be visible to the viewers. If multiple is placed in the tag, the viewers are allowed to select multiple items from the list. Otherwise, only one item can be selected each time by the viewers. The OPTION tag is used to add an item to the list. If an OPTION tag has the SELECTED attribute set, the item is selected in default. Thus, more than one items in the list can have the SELECTED attribute set if the MULTIPLE attribute is set in the SELECT tag and only one item can have the SELECTED attribute set otherwise.

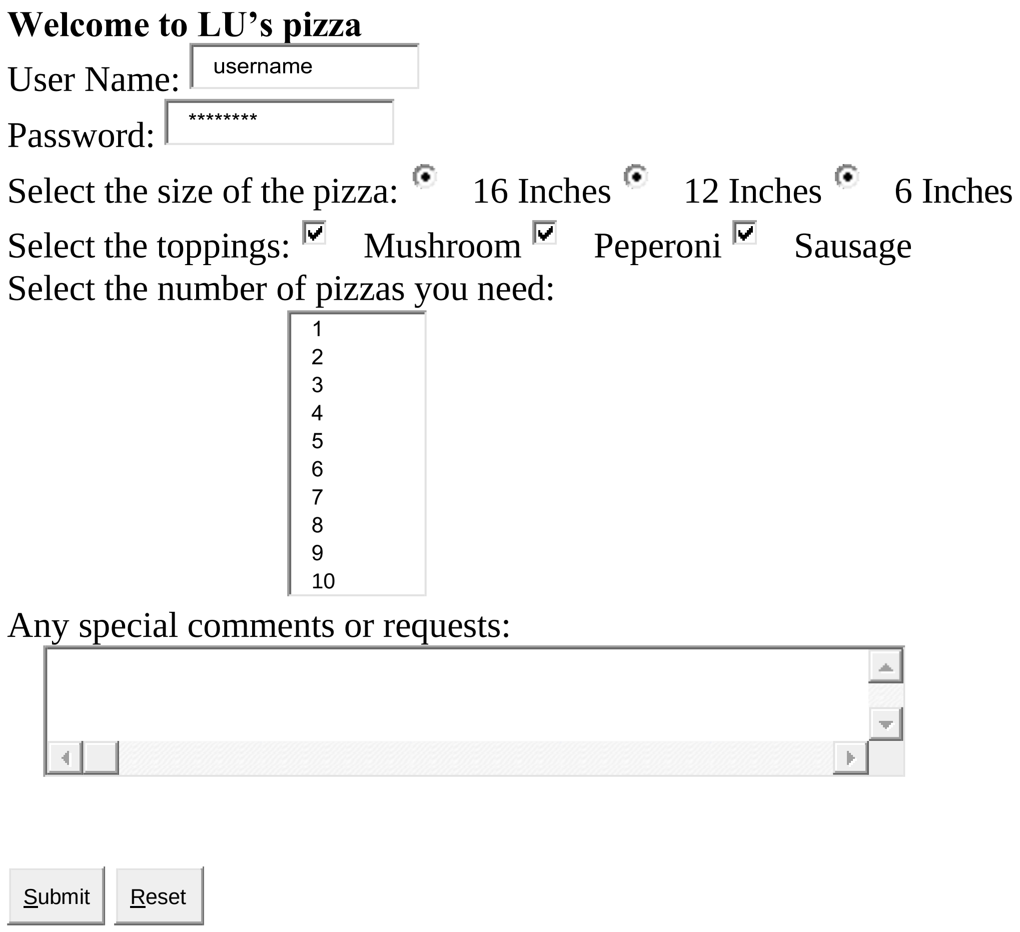
```
<SELECT NAME= “number” size= “10”>
       <OPTION>1</OPTION>
      <OPTION>2</OPTION>
     <OPTION>3</OPTION>
     <OPTION>4</OPTION>
     <OPTION>5</OPTION>
     <OPTION>6</OPTION>
     <OPTION>7</OPTION>
     <OPTION>8</OPTION>
     <OPTION>9</OPTION>
     <OPTION>10</OPTION>

</SELECT>
```

The TEXTAREA element may have the following attributes: name, rows and cols. The name attribute is no diffterent from the other elements. The rows and cols attributes are used to specify the number of rows and columns visible to the viewers. Now we can complete our pizza order form.

```
<FORM ACTION= “serverServlet.java” MEHTOD= “GET”>
```

```
   <STRONG>Welcome to LU’s pizza</STRONG><BR/>
   User Name: <INPUT TYPE=“text” NAME=“username”
VALUE=“user” SIZE= “20”/>
                      <BR/>
   Password:  <INPUT TYPE= “password” NAME= “pw”
VALUE=“password” SIZE=
                     “20”/> <BR/>
   Select the size of the pizza:
                 <INPUT TYPE= “radio” NAME= “size” VALUE=
“16” CHECKED/> 16 Inches
                 <INPUT TYPE= “radio” NAME= “size” VALUE=
“12”/> 12 Inches
                 <INPUT TYPE= “radio” NAME= “size” VALUE=
“6”/> 6 Inches<BR/>
  Select the toppings:
          <INPUT TYPE=“checkbox” NAME=“toppings”
VALUE=“mushroom” CHECKED/>
          Mushroom
         <INPUT TYPE=“checkbox” NAME=“toppings”
VALUE=“pepperoni” CHECKED/>
          Peperoni
          <INPUT TYPE=“checkbox” NAME=“toppings”
VALUE=“sausage” CHECKED/>
          Sausage<BR/>
   Select the number of pizzas you need:<BR/>
   &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp;
          <SELECT NAME= “number” size= “10”>
             <OPTION>1</OPTION>
            <OPTION>2</OPTION>
           <OPTION>3</OPTION>
           <OPTION>4</OPTION>
           <OPTION>5</OPTION>
           <OPTION>6</OPTION>
           <OPTION>7</OPTION>
           <OPTION>8</OPTION>
           <OPTION>9</OPTION>
           <OPTION>10</OPTION>
               </SELECT><BR/>
      Any special comments or requests:<BR/>
      &nbsp;&nbsp;
              <TEXTAREA name= “comment” rows= “3” cols=
“80”>
                      Enter your comment here
              </TEXAREA><BR/><BR/><BR/>
  <INPUT TYPE= “submit” NAME= “Submit”/>
  <INPUT TYPE= “reset” NAME=”Reset”/>
</FORM>
```

We used the HTML entity &nbsp, non breaking spaces, to add some spaces. The display should look like the following.



Welcome to LU’s pizza

User Name: username

Password: ********

Select the size of the pizza: ● 16 Inches ● 12 Inches ● 6 Inches

Select the toppings: ✓ Mushroom ✓ Peperoni ✓ Sausage

Select the number of pizzas you need:

1
2
3
4
5
6
7
8
9
10

Any special comments or requests:

8.2 Cascading Style Sheet (CSS) Basics

CSS, the acronym of cascading style sheet, is a language used to style web pages.
A CSS statement has the syntax: property:value;
The following are some sample properties and corresponding values.

font-family	arial, sans-serif
color	red, yellow, blue, green
font-size	22px, 1in, 1em

To find where we may place CSS statements, we need to learn the three major CSS selectors.

1). HTML selector

An HTML selector is used to define styles associated to a specific HTML element/tag. The general syntax of an HTMP selector is HTMLTagName{CSS statements separated by commas}. Once an HTML selector is defined, every HTML element/tag with the tag name of the selector in the entire page is redefined.

Here is an example page that uses an HTML selector.
```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
       <style type= "text/css">
         p{font-family: arial; font-size:14px; color:red;}
       </style>
  </HEAD>
  <BODY>
        <p>The style of this paragraph and the text enclosed
           in all other p tags are redefined by the css code
           defined in the head section of this page</p>
  </BODY>
</HTML>
```
2). Class selector

A class selector is used to define styles that can be used without redefining HTML tags. The general syntax of a class selector is .className{CSS statements separated by commas}. As you may have noticed, the name of a class selector must be preceded by a dot symbol. If the content of an HTML element/tag needs to be styled by a class selector, the name of the class selector should be assigned to the class attribute of the element/tag. Here is a simple page that uses a class selector for styling a paragraph.
```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
       <style type= "text/css">
         .headline{font-size:14px; color:red;}
       </style>
  </HEAD>
  <BODY>
        <p class= "headline">Only the style of this
         paragraph is redefined by the css code defined
         headline selector</p>
  </BODY>
</HTML>
```
3). ID selector

An ID selector is used when you want to define a style relating to an object with a unique id. So an ID selector is used to target a specific element/tag. Recall that an ID is unique and is used to mark a point in a page. Each element can have only one id and each page can have only one element with that id. The syntax of an ID selector is #IDValue{CSS statements separated by commas}. The name of an ID selector must preceeded with the number sign symbol # and the selector name is assigned to the id attribute of an element/tag which uses the style defined by the selector.
```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
       <style type=  "text/css">
         #layer1{position:absolute; left:100px; top:100px;
                  z-index:1px;}
```

```
      </style>
  </HEAD>
  <BODY>
        <div id= "layer1">
          <table border= '1'>
              <tr>
                 <td>This is layer 1 positioned at
                     (100, 100)
                 </td>
              </tr>
          </table>
        </div>
  </BODY>
</HTML>
```

There are two important "dummy" HTML tags that don't do anything in themselves so they are good to be used to carry css styles.
1). The SPAN tag is an "inline tag", meaning that no line breaks are inserted before and after the use of the tag.
2). The DIV tag is a "block tag", meaning that line breaks are automatically inserted to distance the block from the surrounding context so it is good to be used to make boxes.

Where should we place the CSS statements and selectors and which part of a page can be affected by these CSS statements and selectors? To answer these questions, we need to study the three levels of CSS sheet: inline style, embedded style and linked external style.
1). Inline style
An inline style sheet is assigned to the style attribute of an HTML tag as value to declare an individual element's format and override any other styles of the tag. Here is an HTML tag with an inline style sheet:
```
<p style= "font-size:28px; color:red;">This is the content</p>
```

2). Embedded style
An embedded style sheet is placed in an HTML document's head section by using a style element/tag. In fact, we have used embedded styles to illustrate the three kinds of selectors earlier.
3). Linked External Style
You may define all selectors in a separate document, save it as a .css file and use the link tag/element in the head section of an HTML document to link the document to the style sheet. The link element/tag has three attributes: rel, type and href. The value of the rel attribute should be "stylesheet", the value of the type attribute should be "text/css" and the value of the href attribute should be the file path of the css document:
```
<link rel="stylesheet" type= "text/css" href= "stylesfile.css"/>
```
To practice, you may use the following three basic properties and their values.

Property Name	Values
font-weight	bold, normal, bolder, lighter, 700, 400, 900, 100

font-family	arial, helvetica, serif sans-serif, cursive, fantasy, monospace
font-size	1.5em, 24px, 150%, xx-small, x-small small, smaller, medium, large, larger.

For more properties and their values, you may look at the tutorial in http://www.w3schools.com/cssref/.

8.3 CGI vs. Servlets

In the previous section, we studied HTML which can be used to structure the content of web pages. In this dynamic world, Web pages that can interact with the viewers are needed. Before servlet was used, CGI, the acronym of Common Gateway Interface, was the standard method used to generate dynamic content on Web pages and Web applications. CGI provides interface between Web server and programs that generate Web content. CGI programs or CGI scripts are usually written in scripting languages such as Lua, Perl and PHP. Each Web server runs server software to respond to the requests from Web browsers by finding a pre-written file from a directory or document collection. CGI extends this system by allowing the owner of the Web server to place executable scripts instead of pre-written pages into the document collection. Thus, CGI, instead of simply sending a file to a Web browser, runs the script and passes the output of the script to the browser. One disadvantage of CGI is that it spawns a new process every time it is executed, consuming more and more processing time and sever resources every time it is invoked. CGI scripts are also platform dependent and cannot take advantages of a server's capabilities (cannot write to a server's log file) because they run in processes separate from the server's process.

A Java servlet is designed to overcome these disadvantages of CGI. Servlets are efficient and scalable because they don't create new process every time they are executed; instead, servlets are handled by separate threads within Web server process. They can write to server log files and can take advantages of other server capabilities because they run in the server's own process.

A servlet is a Java class that implements the Servlet interface in the javax.servlet package. The Servlet interface has five methods: destroy(), getServletConfig(), getServletInfo(), init(ServletConfig config) and service(ServletRequest req, ServletResponse res). As the rule states, all its methods must be implemented when an interface is implemented. To simplify the work of the programmers, the Java library has two classes, GenericServlet in javax.servlet and HttpServlet in javax.servlet.http, that implement the Servlet interface. The GenericServlet class is protocol independent so it can be extended for using any protocol. Since almost all servlets written today are designed to use the HTTP protocol, most servlets extend the HttpServlet class. Theoretically, a servlet must implement the init() method from the Servlet interface, but a servlet extending the HttpServlet class inherits its init() method so this part can be saved.

8.4 Defining and Activating a Servlet

A servlet is to be used to design a dynamic Web page. It receives data from viewers, processes it and sends the output back to the viewers. One of the most common ways to invoke a servlet from an HTML page is to provide a link to it from a button on a form. A servlet can also be invoked by embedding a link to it in an HTML page or by typing its pathname into the address bar of a browser. The two most important attributes of the FORM tag in an HTML form are ACTION and METHOD. The value of the ACTION attribute is a url specifying the location of a server such as a servlet. The METHOD attribute takes one of the two values: GET or POST.

The two most important methods in the HttpServlet class are doGet and doPost and one of them must be overridden when a servlet extends this class.
protected void doGet(HttpServletRequest req, HttpServletResponse resp)—
    Automatically called by the server via the service method to allow a servlet to handle a GET request
protected void doPost(HttpServletRequest req, HttpServletResponse resp)—
    Automatically called by the server via the service method to allow a servlet to handle a POST request

When the submit button in a form is clicked, the server program specified by the value of the ACTION attribute in the FORM tag is activated and the data in the form is sent to the server program by using the method specified by the value of the METHOD attribute of the FORM tag. If the value of the METHOD attribute is GET, the data is appended to the end of the url, sent to the servlet and the doGet method of the servlet is activated. If the value of the METHOD attribute is POST, the data is packaged and sent to the servlet, and the doPost method of the servlet is activated. Since GET requests can be cached and remain in the browser history, it saves time if the data is used in multiple times. But GET requests send data via url by appending query strings with name-value pairs so they are not secure and have length restriction.

Now, how does a servlet work? A user requests some information by filling out a form containing a link to a servlet and clicking the Submit button; the server locates the requested servlet; then the servlet gathers the data needed to satisfy the user's request and constructs a Web page containing the information; finally that Web page is displayed on the user's browser. So a programmer firdt needs to know how a servlet gathers data and constructs a Web page.

When a servlet is activated by an HTML form, an object of HttpServletRequest and an object of HttpServletResponse are automatically supplied to its doGet or doPost method and one of these methods is activated according to the value of the METHOD attribute in the form tag. A servlet gathers data by using the methods from the HttpServletRequest class and sends the response back to the user by using the methods from the HttpServletResponse class. Both of these classes are in the javax.servlet.http package.

We need three methods from the HttpServletRequest class:
public String getParameter(String name)—Returns the value of a request parameter as a string or null if the parameter does not exist.
public String[] getParameterValues(String name)—Returns an array of String objects containing all of the values the given request parameter has or null if the parameter does not exist.
public RequestDispatcher getRequestDispatcher(String filePath)—Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path. We usually use the returned object to call its forward(ServletRequest req, ServletResponse resp) method to forward a request from a servlet to another resource such as a servlet, JSP file or an HTML file, or call its include(ServletRequest req, ServletResponse resp) method to include the content of a resource such as a servlet, JSP page or an HTML file in the response.

We also need two methods from the HttpServletResponse class:
public PrintWriter getWriter() throws IOException—Returns a PrintWriter object that can be used to send character text to the client.
public void setContentType(String type)—Sets the content type of the response being sent to the client. The given content type may include a character encoding specification such as text/html, text/plain, image/gif and charset=UTF-8.

Now, we are ready to show an example of a servlet: Design a simple Web site that allows the users to use it to order pizza.

First, we design an HTML form through which the users can enter the data of the order.

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
      <TITLE>LU’S PIZZA</TITLE>
  </HEAD>
  <BODY>
      <FORM ACTION= “PizzaOrder” MEHTOD= “POST”>
          <STRONG>Welcome to LU’s pizza</STRONG><BR/>
        User Name:
        <INPUT TYPE=“text” NAME=“username” VALUE=“user”
                           SIZE= “20”/> <BR/>
         Password:
         <INPUT TYPE= “password” NAME= “pw” VALUE=“password”
                           SIZE= “20”/>
        <BR/>
         Please select the size of YOUR pizza:
             <INPUT TYPE= “radio” NAME= “size” VALUE= “16”
                                   CHECKED/> 16 Inches
             <INPUT TYPE= “radio” NAME= “size” VALUE= “12”/>
                                     12 Inches
             <INPUT TYPE= “radio” NAME= “size” VALUE= “6”/>
                                   6 Inches<BR/>
        Select the toppings:
         <INPUT TYPE=“checkbox” NAME=“toppings”
                  VALUE=“mushroom” CHECKED/>Mushroom
         <INPUT TYPE=“checkbox” NAME=“toppings”
                 VALUE=“pepperoni” CHECKED/>Peperoni
         <INPUT TYPE=“checkbox” NAME=“toppings”
                 VALUE=“sausage” CHECKED/>Sausage<BR/>
         Please select the number of pizzas you need:<BR/>
         &nbsp;&nbsp; &nbsp;&nbsp; &nbsp;&nbsp;
                <SELECT NAME= “number” size= “10”>
                   <OPTION>1</OPTION>
                  <OPTION>2</OPTION>
                 <OPTION>3</OPTION>
                 <OPTION>4</OPTION>
                 <OPTION>5</OPTION>
                 <OPTION>6</OPTION>
                 <OPTION>7</OPTION>
                 <OPTION>8</OPTION>
                 <OPTION>9</OPTION>
                 <OPTION>10</OPTION>
                     </SELECT><BR/>
            Any special comments or requests:<BR/>
            &nbsp;&nbsp;
             <TEXTAREA name= “comment” rows= “3” cols= “80”>
                           Enter your comment here
             </TEXAREA><BR/><BR/><BR/>
        <INPUT TYPE= “submit” NAME= “Submit”/>
        <INPUT TYPE= “reset” NAME=”Reset”/>
      </FORM>
     </BODY>
</HTML>
```

A simple servlet that takes the orders and sends response back would be like the following one.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
@WebServlet(urlPatterns = {"/PizzaOrder"})
public class PizzaOrder extends HttpServlet
{
   protected void processRequest(HttpServletRequest request,
              HttpServletResponse response)
              throws ServletException, IOException
   {
      response.setContentType("text/html");
      String user=request.getParameter("user");
      String size=request.getParameter("size");
      String[] toppings=request.getParameterValues("toppings");
      String number=request.getParameter(“number”);
      PrintWriter out = response.getWriter();
      try {
         out.println(“<!DOCTYPE HTML>”);
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Your Order</title>");
         out.println("</head>");
         out.println("<body bgcolor=\“pink\”>");
         out.println("<h1>Hello, " + user + "</h1>");
         out.println("You have ordered ”+number+ “ pizzas of size "+size
                       +" with toppings: ");
         out.println(“<OL>”);
         for(int i=0; i<toppings.length; i++)
            out.print(“<LI>”+toppings[i]+"</LI>");
         out.println(“</OL>”);

         out.println("</body>");
         out.println("</html>");
      } finally {
          out.flush();
          out.close();
      }
   }
   /**Override doGet method that activates the processRequest method of
    *the class
    * @param request servlet request
    * @param response servlet response
    * @throws ServletException if a servlet-specific error occurs
    * @throws IOException if an I/O error occurs
    */
   @Override
   protected void doGet(HttpServletRequest request,
                            HttpServletResponse response)
          throws ServletException, IOException
   {
      processRequest(request, response);
   }
   /**


    * Override the doPost method that activates the processRequest
    *method of the class
    * @param request servlet request
    * @param response servlet response
    * @throws ServletException if a servlet-specific error occurs
    * @throws IOException if an I/O error occurs
    */
   @Override
   protected void doPost(HttpServletRequest request,
HttpServletResponse response)
          throws ServletException, IOException
   {
      processRequest(request, response);
   }

   /**
    * Returns a short description of the servlet.
    * @return a String containing servlet description
    */
   @Override
   public String getServletInfo()
    {
      return "Short description";
   }// </editor-fold>
}
```

In this servlet, we first defined a processRequest method and ask both of the doGet and doPost method to call this method. This makes the servlet suitable for a form that uses either the GET method or the POST method to make request. Since the user, size and number parameters have a single value each in the form, we use the getParameter method from the HttpServletRequest class in this servlet to get their values. The toppings parameter in the form represents a group of checkboxes and may have multiple values so we have to use the getParameterValues method from the HttpServletRequest class to get all the values and store them in a String array. A servlet is a class so it can do anything a regular class can do. Especially, it can process the data received from a form by doing calculation. Both the getParameter and getParameterValues methods return String type values and they must be parsed into number types by using the parseInt method from the Integer class or parseDouble method from the Double class if calculations are needed. Don’t forget the greatest advantage of the Java language: object-orientation. Design the servlet with multiple classes so that the program is robust and scalable.

By looking at this example, one should have realized the disadvantage of using a servlet to design a dynamic Web site. A programmer needs to embed HTML code into Java code which needs the programmer to be good at both Java programming and Web design. A minor disadvantage is that the mixture of Java code with HTML code. This second disadvantage may be reduced to the minimum by putting the code into separate methods or even classes.

8.5 Session Tracking

HTTP protocol is stateless so an HTTP Web server generally cannot associate request from a client and therefore it treats each request independently. If your Web site is for selling products, and a customer selects some items and moves to another page for getting some other information, the selection is lost. To overcome this problem, Java allows two ways for session tracking that maintains session between Web client and Web server. Another important use of session tracking is to allow multiple pages in a session to share data.

A simple way of session tracking is using cookies. A cookie is a small text file that stores sets of name-value pairs on the disk in the client machine. Cookies are sent from the server through the instructions in the header of the HTTP response. The instructions tell the browser to create a Cookie with a given name and its associated value. If the browser already has a Cookie with the given name, the value will be updated. The browser will then send the Cookies with any request submitted to the same server. To accomplish these tasks, we first need to understand the Cookie class from the javax.servlet.http package.

Cookie(String name, String value)—Constructs a cookie object with the specified name and value.
String getName()—Returns the name of the Cookie.
String getValue()—Returns the value of the Cookie.
void setValue(String newValue)—Assigns the given value to the Cookie after it is created.

To send a cookie to a browser, we need the addCookie method from the HttpServletResponse class:
void addCookie(Cookie cookie)—Adds the specified cookie to the response.

To obtain cookies from a browser, we need to use the following method from the HttpServletRequest class:
Cookie[] getCookies()—Returns an array containing all the cookie objects the client sent with this request.

Now we design a simple Web site that is used to order books for illustrating how cookie session tracking is used.

```
<!DOCTYPE HTML>
<HTML>
```

```
  <HEAD>
      <TITLE>LU’S BOOKSTORE</TITLE>
      <STYLE type= “text/css”>
          body{background-color:pink;}
      </STYLE>
  </HEAD>
  <BODY>
     <FORM ACTION= “BookOrder” MEHTOD= “POST”>
         <STRONG>Welcome to LU’s Bookstore</STRONG><BR/>
       User Name:
       <INPUT TYPE=“text” NAME=“username” VALUE=“user”
                     SIZE= “20”/> <BR/>
        Password:
        <INPUT TYPE= “password” NAME= “pw” VALUE=“password”
                    SIZE= “20”/>
       <BR/>
        Please select the books you want to buy:
        <INPUT TYPE=“checkbox” NAME=“books” VALUE=“java1”
                   CHECKED/>
               Object-oriented Programming with Java</BR>
         <INPUT TYPE=“checkbox” NAME=“books” VALUE=“java2”
              CHECKED/>
              Graphic User Interface with Java</BR>
         <INPUT TYPE=“checkbox” NAME=“books” VALUE=“java3”
              CHECKED/>Advanced Java<BR/>
         <INPUT TYPE=“checkbox” NAME=“books” VALUE=“df”
              CHECKED/>
              Ordinary Differential Equations<BR/>
        <INPUT TYPE=“checkbox” NAME=“books” VALUE=“web”
              CHECKED/>Internet and Web Programming<BR/>

           Any special comments or requests:<BR/>
           &nbsp;&nbsp;
            <TEXTAREA name= “comment” rows= “3” cols= “80”>
                           Enter your comment here
            </TEXAREA><BR/><BR/><BR/>
        <INPUT TYPE= “submit” NAME= “Submit”/>
        <INPUT TYPE= “reset” NAME=”Reset”/>
     </FORM>
    </BODY>
</HTML>
```

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
@WebServlet(urlPatterns = {"/BookOrder"})
public class BookOrder extends HttpServlet
{
   protected void processRequest(HttpServletRequest request,
               HttpServletResponse response) throws ServletException,
IOException
   {
      response.setContentType("text/html");
      String user=request.getParameter("user");
      String[] books=request.getParameterValues("books");
      PrintWriter out=response.getWriter();
      Cookie[] cookies=request.getCookies();
      int n=((cookies==null)?0:cookies.length);
      int m=books.length;
      Cookie[] newCookies=new Cookie[m];
       for (int i=0; i<books.length; i++)
      {
        //This is the cookie number i+n
        newCookies[i]=new Cookie("book"+(i+n), books[i]);
        response.addCookie(newCookies[i]);
      }
      try {
         out.println("<!DOCTYPE HTML>");
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Your Book Order</title>");
         out.println("</head>");
         out.println("<body bgcolor=\"#ffffff\">");
         out.println("<h1>Hello, " + user + "</h1>");
         out.println("Your current order is ");
         out.println("<OL>");
         if(cookies!=null)
           for(int i=0; i<cookies.length; i++)
                out.print("<LI>"+cookies[i].getValue()+"</LI>");
        for(int i=0; i<books.length; i++)
        {
           response.addCookie(new Cookie("book"+(i+n), books[i]));
           out.print("<LI>"+books[i]+"</LI>");
        }
         out.println("</OL>");

         out.println("</body>");
         out.println("</html>");
      } finally {
          out.flush();
             out.close();
      }
   }
   /**Override doGet method that activates the processRequest method of
    *the class
    * @param request servlet request
    * @param response servlet response
    * @throws ServletException if a servlet-specific error occurs
    * @throws IOException if an I/O error occurs
    */
   @Override
   protected void doGet(HttpServletRequest request,
                                 HttpServletResponse response)
                                   throws ServletException, IOException
   {
      processRequest(request, response);
   }
   /**

    * Override the doPost method that activates the processRequest
    *method of the class
    * @param request servlet request
    * @param response servlet response
    * @throws ServletException if a servlet-specific error occurs
    * @throws IOException if an I/O error occurs
    */
   @Override
   protected void doPost(HttpServletRequest request,
                                HttpServletResponse response)
         throws ServletException, IOException
   {
      processRequest(request, response);
   }

   /**
    * Returns a short description of the servlet.
    * @return a String containing servlet description
    */
   @Override
   public String getServletInfo()
    {
      return "Short description";
   }// </editor-fold>
}
```

The example shows that session tracking with cookies does work well, but we need to realize that it has several shortcomings. First, this method depends on whether the client machine accepts cookies or not. If the client machine does not

accept cookies, this method would not work. Second, it can only store small amount of data because a cookie is a small text file of name-value pair. Third, the data is not secure because cookies are stored in the client machine. A second method of session tracking in Java is to use the servlet API. The HttpSession interface in the Java library provides a way to identify a user across more than one page request and to store a user. The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session enables tracking of a large amount of data and makes it secure by storing the data in a container such as ArrayList and Vector.
To use the Java servlet API session tracking, you need to get an HttpSession object by activating the getSession() method of the HttpServletRequest class:
HttpSession getSession()—Returns the session associated with this client or creates a new session if the client does not have a session on this server.
Then, you may use the following methods of the HttpSession class to strore data:
Object getAttribute(String name)—Returns the object bound with the specified name in this session or null if no object is bound with this name.
void setAttribute(String name, Object value)—Binds an object to this session, using the specified name. If an object of the same name is bound to this session already, the object is replaced.
void removeAttribute(String name)—Removes the object bound with the specified name from this session.
As you have noticed that the return type of getAttribute and the second parameter o the setAttribute methods are all Object type. You may bind a String type name with any Object by using the setAttribute method.
When you use the getAttribute method to get the Object back, you need to cast it back to its actual type.
Now, we can modify the previous servlet so it uses servlet API session tracking.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
@WebServlet(urlPatterns = {"/BookOrder"})
public class BookOrder extends HttpServlet
{
   protected void processRequest(HttpServletRequest request,
              HttpServletResponse response) throws ServletException,
              IOException
   {
      response.setContentType("text/html");
      String user=request.getParameter("user");
      String[] books=request.getParameterValues("books");
      PrintWriter out=response.getWriter();
      HttpSession session=request.getSession()
   ArrayList<String> items
                 =(ArrayList<String>)session.getAttribute(“items”);
    if(items==null) items=new ArrayList<String>();
    for (int i=0; i<books.size(); i++)
       items.add(books.get(i)); //add the new order into the object
     session.setAttribute(“items”, items)

    try {
       out.println(“<!DOCTYPE HTML>”);
       out.println("<html>");
       out.println("<head>");
       out.println("<title>Your Book Order</title>");
       out.println("</head>");
       out.println("<body bgcolor=\“#ffffff\”>");
       out.println("<h1>Hello, " + user + "</h1>");
       out.println("Your current order is ");
       out.println(“<OL>”);
       if(items!=null)
         for(int i=0; i<items.size(); i++)
              out.print(“<LI>”+items.get(i)+"</LI>");
       out.println(“</OL>”);
       out.println("</body>");
       out.println("</html>");
    } finally {
        out.flush();
           out.close();
    }
}
/**Override doGet method that activates the processRequest method of
 *the class
 *@param request servlet request
 *@param response servlet response
 *@throws ServletException if a servlet-specific error occurs
 *@throws IOException if an I/O error occurs
 */
@Override
protected void doGet(HttpServletRequest request,
                                     HttpServletResponse response)
                                  throws ServletException, IOException
{
    processRequest(request, response);
}
/**

 * Override the doPost method that activates the processRequest
 *method of the class
 * @param request servlet request
```

```
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
                          throws ServletException, IOException
    {
        processRequest(request, response);
    }

    /**
     * Returns a short description of the servlet.
     * @return a String containing servlet description
     */
    @Override
    public String getServletInfo()
     {
        return "Short description";
    }// </editor-fold>
}
```

A website usually has many pages linked with each other. A servlet can collect the information from the client and use the data collected to direct the user to another page by sending out another form.

Review Questions

1. What is HTML?
2. Describe the general syntax of HTML.
3. What are the two major parts of an HTML file?
4. What is the tag or element used to change line in HTML?
5. What is the HTML element/tag including attributes for inserting an image into a page?
6. What is the HTML element/tag including attributes for connecting your page to an existing page?
7. Describe the tags or elements used for three types of lists in HTML?
8. What are the two attributes in a form tag we have studied? What are their possible values?
9. In an INPUT element of a form, there is a name attribute. What is the value of a name attribute for?
10. In an INPUT element of a form, there is a value attribute, what is the value of this attribute for?

11. The method attribute in a FORM element/tag may take one of the following two values: GET, POST. What is the effect of using each of these values?
12. What is a Servlet?
13. Describe the getPameter method from the HttpServletRequest class carefully.
14. Describe the getWriter method from the HttpServletResponse class carefully.
15. What is CGI? How does it work?
16. What is a Servlet? What is the advantage of a Servlet over CGI?
17. How is a Servlet activated?
18. The HttpServlet class has a doGet method and a doPost method? How are these methods activated?
19. What are the methods that a Servlet uses to get data from a form activating it?
20. How does a Servlet send output back to the users?
21. What is a Cookie?
22. What are the disadvantages of using Cookies for session tracking?
23. How to send a Cookie to a browser?
24. How to get the Cookies from a browser?
25. How does the getSession() method from the HttpServletRequest class work?
26. How to bind an object to a session?

Programming Projects

1. Design a **website** and a **database** by using Servlets, HTML forms, JDBC and regular classes to support electronic banking. First, you need to have a database that contains the information of the customers. This database needs to be well designed with appropriate table structures that contain all customer information including customers' id, User Name, Password, name, address, the balance of each of the three accounts (Saving, Checking, and Money Market) of each customer if s/he has any, and the transaction information on each account made by each customer. Your webpage will allow the customers to open a new account, deposit to or withdraw from an account, transfer from one account to another and list all transactions made to an account within a certain time period supplied by the customer. When a customer first opens your webpage, it should ask the customer if s/he wants to login or open a new account. If a customer wants to open a new account, you should ask the customer to enter his/her name address, a user name, a pass word, what kind(s) of account(s) to open and how much to deposit into each new account. Of course, you should store all information supplied by the customer into your database. If a customer wants to deposit some money, your webpage should show a list that includes all the accounts the customer has so s/he can select an account depositing into. If a customer wants to withdraw, your webpage should show the list of accounts

the customer has and the corresponding balance in each account so that your customer may select an account to withdraw from. If a customer wants to look at the transactions within a period, you should ask the customer to select an account, enter the beginning date and ending date of the period. If a customer wants to transfer money from one account to another, your webpage should show the list of the accounts with the balances that the customer has and allow the customer to select the one s/he wants to transfer from, then show the list again for the customer to select the one s/he wants to transfer to.

2. Use Servlet and JDBC to design the WeChat program specified in Chapter seven.

Chapter Nine
Java Server Pages (JSP)

Servlets provide a way to generate dynamic Web pages that are faster to run. A programmer may use all the powerful features of Java such as exception handling and garbage collection when a servlet is designed. Since Java is portable across different platforms, servlet enables easy portability across Web servers. The drawback is that Servlets embed HTML code into the Java code so the programmer needs to be skillful in both Java and HTML programming. With HTML code embedded in Java code, it also makes the document look ugly. To solve this problem, we can use a mixture of both servlets and JSP.

A JSP page is a text document that contains regular static HTML code in the normal way and embeds Java code to produce dynamic contents. The extension of the source file of a JSP page is .jsp. For understanding how to code a JSP page, it is important to know how a JSP page is processed. Once a browser sends an HTTP request to a Web server, the Web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. The JSP engine loads the JSP page from disk and converts it into a servlet content so all template text is converted to println() statements and all JSP elements are converted to Java code that implements the corresponding dynamic behavior of the page. Finally, the JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine. So in a way, a JSP page is really another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled like a regular servlet.

Again, A JSP page may contain regular static HTML code and JSP elements. There are three types of JSP elements: scripting constructs, directives and actions.

9.1 Scripting Constructs and Comments

Scripting constructs are used to insert Java code that will become part of the resultant servlet. There are three main types of scripting constructs: expression, scriptlet and declaration.

A JSP expression is used to insert a Java expression directly into the output and has the following syntax:

```
  <%= Java expression %>
```

Recall that a Java expression consists of digits, variables, operands and method calls.

A JSP scriptlet is used to insert a Java statement into the resultant servlet's service method and has the following syntax:

```
<% Java statement %>
```

A JSP declaration is for declaring methods or fields into the servlet and has the following syntax:

```
<%! Java declaration %>
```

Clearly, you can also declare variables and methods in a JSP scriptlet. The difference is that a variable declared in a JSP declaration becomes an attribute of the resultant servlet and it is initialized automatically with its default value if it is not assigned a value at declaration. A variable declared in a JSP scriptlet becomes a local variable of the service method of the resultant servlet.

Since a JSP page may contain regular static HTML code, you may use the regular HTML comment <!—HTML comments --> in a JSP page directly. If you don't want the comment to appear in the resultant HTML file, you may use JSP comment in the following syntax:

```
<%-- JSP comment --%>
```

Now, we can design a simple JSP page that calculates and displays the factorials of integers 0 through 10.

```
<!DOCTYPE html>
<HTML>
  <HEAD>
   <TITLE>Factorial Calculator</TITLE>
   <STYLE type= “text/css”>
      body{background-color:#FFFF00;}
   </STYLE>
  </HEAD>
  <BODY>
     <STRONG>Here are the factorials of the integers 0
             through 10:
     </STRONG>
     <% for(int i=0; i<=10; i++)
         {
      %>
          Factorial of <%= i %> is
      <%= factorial(i) %> </BR>
      <% } %>
      <%! private long factorial(int n)
          {
            if (n==0) return 1;
            else return n*factorial(n-1);
          }
      %>
  </BODY>
</HTML>
```

It is helpful to think about what the resultant servlet looks like. Recall that the regular static HTML text is converted to println() statements and all JSP elements are converted to Java statements. The JSP declaration

```
 <%! private long factorial(int n)
   {…}
  %>
```

should be converted to an instance method of the servlet. The rest in the body should be converted to the following Java statements in the body of the service method:

```
out.println("Here are the factorials of the integers "+ "0
              through 10:");
  for(int i=0; i<=10; i++)
  {
    out.println("Factorial of "+i+" is "
                +factorial(i));
    }
```

In this translation, out is a predefined variable representing a PrintWriter object that will be covered in next section.

9.2 Predefined Variables

In a servlet, we need to use the methods from the HttpServletRequest and HttpServletResponse classes. Since a JSP page is converted to a servlet, these methods are needed, but the HttpServletRequest and HttpServletResponse classes don't directly show up in a JSP document. To solve this problem, the JSP language provides several predefined variables or implicit objects.

request—A variable represents an object of the HttpServletRequest class so it can be used in a JSP to activate the methods of this class.
response—A variable represents an object of the HttpServletResponse class so it can be used in a JSP to activate the method of this class.
out—A variable represents an object of the PrintWriter class and can be used to send output to the client. Since JSP expressions automatically get placed in the output stream, out is rarely needed explicitly.

session—A variable represents the HttpSession object associated with the request.

Now, we can use JSP to design a simple Web Site that asks the users to enter the information about their mortgage, calculates the monthly payments and reports back to the users.

```
<!—Mortgage.html
  Ask the user to enter his/her mortgage information,
calculates and reports back the monthly payment.
-->
<!DOCTYPE HTML>
<HTML>
    <HEAD>
        <TITLE>Mortgage Calculator</TITLE>
        <STYLE type=”text/css”>
            body{background-color:pink;}
        </STYLE>
    </HEAD>
    <BODY>
        <STRONG>Enter the info about your mortgage, I will
               Calculate and give you your monthly payment:
        </STRONG>
        <FORM ACTION="MortgageJSP.jsp" METHOD="post"><BR/>
         Principal: <INPUT TYPE="text" NAME="principal"
                 VALUE="200000"  SIZE="6"/><BR/>
         Annual Interest Rate:<INPUT TYPE="text"
                NAME="rate" VALUE="5.25" SIZE="4"/>%<BR/>
         Life of Your Mortgage in Years:
             <INPUT TYPE="text" NAME="years" VALUE="15"
                SIZE="2"/><BR/>
             <INPUT TYPE="submit" VALUE="Submit"/>
             <INPUT TYPE="reset" VALUE="Set Back"/>
       </FORM>
    </BODY>
</HTML>
```

```
<%--Mortgage.jsp-->

<HTML>
  <HEAD>
   <TITLE>Mortgage Calculator</TITLE>
  </HEAD>
  <BODY BGCOLOR= “#FFFF00”>
     <% double p =
    Double.parseDouble(request.getParameter(“principal”));
       double annualRate =
    Doulbe.parseDouble(request.getParameter(“rate”));
     int life=
     Integer.parseInt(request.getAParameter("years"));
      %>
     <STRONG>Here are all the information of your mortgage:
     </STRONG>
     <STRONG>Principal: </STRONG> <%= p %> <BR/>
     <STRONG>Annual Interest Rate:</STRONG> <%=
          annualRate %>%</BR>
     <STRONG>Life in Years:</STRONG> <%= life %></BR>
     <STRONG> Monthly Payment:</STRONG>
            <%= mpt(p, annualRate, life) %></BR>
     <FORM ACTION= "Mortgage.html" METHD="GET">
        <STRONG> If you need to do another calculation,
                please click the following button
        </STRONG>
        <INPUT TYPE="submit" VALUE="Calculate"/>
     </FORM>
      <%! private double mpt(double p, double rate, int
                               years)
          {
            double mRate=rate/12/100;
            return p*mRate/(1-Math.pow(1+mRate,
                              -12*years));
          }
       %>
  </BODY>
</HTML>
```

9.3 Directives and Exception Handling

A JSP directive gives information and special instructions to container for translation of the JSP to servlet code. A JSP directive has the syntax:
<%@directive_name attribute1="value1" attribute2="value2" %>.

JSP provides three types of directives: page directive, include directive and taglib directive.

A page directive may have quite a few attributes. Among them are the following ones.
1). import—specifies one or more packages to be imported for this page. For example,
    <%@page import= "java.util.*, java.text.*" %>

2). errorPage—specifies a JSP page that is processed when an exception occurs in the current page. For example, <%@page errorPage="HandleError.jsp" %> specifies that HandleError.jsp is processed when an exception occurs in this page.
3). isErrorPage—specifies a boolean value to indicate whether the page can be used as an error page. More precisely, it is made available an object with variable name "exception" of the Throwable class so we may use the methods of this object to display useful information to the users. Now, you should add the directive tag
<%@page errorPage="HandleError.jsp" %>
to the top of the current page and place the directive tag
<%@page isErrorPage= "true" %> on the top of the page HandleError.jsp.
An include directive lets you insert a file to the result servlet when the page is translated. One attribute of the include directive is file and its value is used to specify the file path of the file that is inserted.
Before we look at the third type of directives, let's first modify our example in the previous section to illustrate how the first two directives are used. In the example of the previous section, we didn't care about the format of the output of the numbers, nor the exceptions that may be caused by non-numerical input from the users. To take care of these problems, we need to use the methods from the NumberFormat class and an exception handler.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page isErrorPage="true" %>
<!DOCTYPE HTML>
<HTML>
    <HEAD>
        <TITLE>Exception Handler</TITLE>
        <STYLE type="text/css">
          Body{background-color:blue;}
        </STYLE>
    </HEAD>
    <BODY>
        <STRONG>You have an exception:</STRONG>
        <%=exception.getMessage()%>
        <%@include file= "Mortgage.html" %>
    </BODY>
</HTML>
```

In this simple page, we used the isErrorPage attribute in a page directive to indicate this page is used to handle exceptions so that the predefined variable exception is valid. If you compare the try-catch block used to handle exceptions in regular java code, the page that uses this page to handle exception is corresponding to the try block, this page is corresponding to the catch block, the predefined variable exception behaves like the parameter in the parentheses of the catch block and it contains the information of the exception thrown in the page which uses the attribute errorPage of the page directive to connect to this page. One caveat is that Intenet Explorer may need extra configuration for using this exception handling functionality.

```
<%--Document: Mortgage.jsp-->
<!DOCTYPE HTML>
<%@PAGE IMPORT= "java.text.*" %>
<%@PAGE errorPage= "/ExceptionHandler.jsp">
<HTML>
  <HEAD>
   <TITLE>Mortgage Calculator</TITLE>
    <STYLE type="text/css">
        Body{background-color:#FFFF00;}
    </STYLE>
  </HEAD>
  <BODY>
     <% double p=Double.parseDouble(
               request.getParameter("principal"));
        double annualRate=
          Doulbe.parseDouble(request.getParameter("rate"));
        int life=
          Integer.parseInt(request.getAParameter("years"));
        NumberFormat nf=NumberFormat.getNumberInstance();
        nf.setMaximumFractionDigits(2);
      %>
     <STRONG>Here are all the information of your mortgage:
     </STRONG>
     <STRONG>Principal: </STRONG> <%= nf.format(p) %> <BR/>
     <STRONG>Annual Interest Rate:</STRONG>
                 <%= nf.format(annualRate) %>%</BR>
     <STRONG>Life in Years:</STRONG> <%= life %></BR>
     <STRONG> Monthly Payment:</STRONG>
           <%= nf.format(mpt(p, annualRate, life)) %></BR>
      <%@PAGE INCLUDE= "/Mortgage.HTML"%>
      <%! private double mpt(double p, double rate, int
                             years)
          {
            double mRate=rate/12/100;
            return p*mRate/(1-Math.pow(1+mRate,
                                      -12*years));
          }
       %>
  </BODY>
</HTML>
```

Since the parseInt from the Integer class and the parseDouble method from the Double class may throw NumberFormat excpetion, we added a page directive which uses its errorPage attribute to specify the page for handling exceptions if the code in this page throws any exception.

The third type of JSP directive is called tablib used to declare that your JSP uses a set of custom tags, identify the location of the library and provide a mean for identifying the custom tags in your JSP page. It has two attributes:
uri—specifies the location of the file.
prefix—informs a container what bits of markup are custom actions. The taglib directive follows the following syntax:
<%@TAGLIB URI= “uri” PREFIX= “prefixOfTag” %>.

Since this part requires some understanding of XML, we cover it briefly. You may come back to understand better after we cover XML in next chapter. For example, suppose the custlib tag library contains a tag called hello. If you want to use the hello tag with a prefix mytag, your tag would be <mytag:hello> and it would be used in your JSP file as follows:

```
<%@ TAGLIB URI= “http://www.example.com/custlib”   PREFIX=
“mytag” %>
<HTML>
   <BODY>
      <mytag:hello/>
   </BODY>
</HTML>
```

9.4 JSP Actions

We wanted to use JSP to design web pages because we didn’t like to mix HTML code with Java code. To this point, we have been using JSP to embed Java code into HTML code. To minimize the use of Java code in a JSP page, we need to place Java code in Java classes and use the methods in our JSP page. The JSP actions are for this purpose.

The JSP actions allow different pages to share object of a JavaBean. A JavaBean is a class that satisfies:
1). The class is public;
2). The class has a constructor without parameter or it does not have any constructor so the default constructor is used;
3). (optional)The class has getProperty and setProperty methods for each property.

The first two conditions are required for a Java class to be a JavaBean and they are simple and clear. The third condition is optional, but very important for a JavaBean to be used in a JSP page. Here a property means an attribute of the class. For example, if a JavaBean has an int type attribute with name number, two corresponding methods should be

```
public int getNumber(){ return number;}
public void setNumber(int num){number=num;}
```

Now, we define a JavaBean for the problem in the previous section and then modify it so it uses JSP actions and JavaBean.

```
/**A class that is used to calculate monthly payment of mortgage*/
public class MortgageBean
{
  private int years; //Mortgage life in years
 private double rate; //annual interest rate with % dropped
 private double principal; //amount borrowed
 private String mpmt; //monthly payment
 private NumberFormat nf;

 public MortgageBean()
 {
  years=0; rate=0;
  principal=0;
  nf=NumberFormat.getInstance();
  nf.setMaximumFractionDigits(2);
 }

 public void setYears(int y){years=y;}
 public int getYears(){return years;}

 public void setRate(double r){rate=r;}
 public double getRate(){return rate;}

 public void setPrincipal(double p){principal=p;}
 public double getPrincipal(){return principal;}

 public String getMpmt()
 {
    double mrate=rate/100/12;
    return nf.format(principal*mrate/(1-Math.pow(1+mrate,
                                        -12*years)));
 }
}
```

You certainly can use the class we have defined in a JSP document by creating objects and activating the methods in JSP scriptlets, but the JSP actions make it more convenient for using a Java class in a JSP document. There are quite a few JSP actions and we will look at five of them.

1). useBean—To get the Java bean object from the given scope or to create a new object of Java bean.

The useBean action has three attributes: id, scope and class, and has syntax:

<jsp:useBean id="variableName" scope="scopeValue" class="className"/>.

When this action is processed, the JSP engine first searches for an object of the class with the same id and scope. If it is found, the pre-existing one is used; otherwise, a new bean of the class is created and assigned to the variable defined by the value of the id attribute. There is another syntax for useBean which provides more functionality:

```
<jsp:useBean id="variableName" scope="scopeValue" class="className">
  Statements
</jsp:useBean>
```

With this syntax, if a new bean needs to be created, the statements enclosed in the start and end tags are executed. If a bean with the same id and class name already exists in the given scope, the statements enclosed in the start and end tags are skipped. The scope attribute may take one of the following four values:

application—Specifies that the object is bound to the application so that the object can be shared by all sessions of the application. Thus, if a client closes one request and opens another one, the object from the previous request still exists.

session—Specifies that the object is bound to the client's session. A client's session is automatically created between a Web browser and a Web server. When a client from the same browser accesses two pages on the same server, the session is the same.

page—This is the default scope, which specifies that the object is bound to the page.

request—Specifies that the object is bound to the client's request. One request may be sent to several pages.

2). setProperty—to set the property of a Java bean object, used with useBean action.

The setProperty action has three attributes: name, property and value, and its syntax is as following:

<jsp:setProperty name="beanID" property= "attributeName" value="valueforAttribute"/>

Here the value of the name attribute is the variable name assigned to the ID attribute of the useBean action; the value of the property attribute is an attribute name of the class and the value of the value attribute is the value assigned to the attribute. The value of the property attribute can be *. In this case, the value attribute is not needed and the object receives the values of all inputs from the form activating this page and assigns them to the corresponding attributes of the object. If you use asterisk * for the value of the property attribute, you need to make sure that the name value of each input in the form activating this page is identical to the corresponding attribute name of the object.

3). getProperty—Retrieves the value of the bean property specified by the property attribute, converts it to a string and inserts it to the output.
This action has two attributes: name and property, and has the syntax:
<jsp:getProperty name="beanID" property="attributeName"/>

4). include—To include a resource such as HTML, JSP or any other file at runtime.
This action has two attributes: page and flush, and has the following syntax:
<jsp:include page="url" flush= "true"/>. It is unlike the include directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested.

5). forward—To forward the request to another resource.
This action has only one attribute: page and its value is used to specify the location of the page to which it is forwarded.

Now, we can modify our Mortgage.jsp so that actions are used.

```
<%--Document: Mortgage.jsp-->
<!DOCTYPE HTML>
<%@PAGE IMPORT= "java.text.*" %>
<%@page IMPORT= "MortgageBean" %>
<%@PAGE errorPage= "/ExceptionHandler.jsp">
<HTML>
  <HEAD>
   <TITLE>Mortgage Calculator</TITLE>
   <STYLE type="text/css">
      Body{background-color:#FFFF00;}
   </STYLE>
  </HEAD>
  <BODY>
      <jsp:useBean id="mortgageBean" class=
                                    "MortgageBean"/>
      <jsp:setProperty name= "mortgageBean" property= "*"/>
     <STRONG>Here are all the information of your mortgage:
     </STRONG>
```

```
    <STRONG>Principal: </STRONG>
     <jsp:getProperty name= "mortgageBean" property=
                                     "principal"/>
     </BR>
    <STRONG>Annual Interest Rate:</STRONG>
     <jsp:getProperty name= "mortgageBean"
                           property= "rate"/>BR>
    <STRONG>Life in Years:</STRONG>
     <jsp:getProperty name= "mortgageBean"
                         property= "years"/>BR>
    <STRONG> Monthly Payment:</STRONG>
     <jsp:getProperty name= "mortgageBean"
                              property= "mpmt"/></BR>
     <jsp:include PAGE= "/Mortgage.HTML" />
  </BODY>
</HTML>
```

9.5 A Comprehensive Example

In this section, we combine what we have covered about JSP in this chapter with JDBC and design a Web site for Bloomsburg University. This Web site will allow the staff in BU to look for information about students. **Do not forget the add the connector to the library. May also need to load the driver.**

We first design our JavaBean which has methods needed for connecting to the database supplied by the user and get the names of the table in the database.

```
/*
 * Document: DBBean.java
 */
package newpackage;
import java.sql.*;

public class DBBean
{
   private String driver;
   private String url;
   private String username;
   private String password;
   private Connection connection;


   public void setDriver(String d){driver=d;}
   public String getDriver(){return driver;}
```

```
  public void setUrl(String u){url=u;}
  public String getUrl(){return url;}

  public void setUsername(String u){username=u;}
  public String getUsername(){return username;}

  public void setPassword(String p){password=p;}
  public String getPassword(){return password;}

  public Connection getConnection(){return connection;}

  public void initializeJdbc()
  {
     try
     {
      Class.forName(driver);
      connection=DriverManager.getConnection(url,
                           username, password);
     }catch(Exception ex)
     {
        ex.printStackTrace();
     }
  }

  public String[] getTables()
  {
     String[] tables=null;
     try
     {
      DatabaseMetaData dbmd=connection.getMetaData();
      ResultSet rs=dbmd.getTables(null, null, "%", new
                            String[]{"TABLE"});
      int size=0;
      while(rs.next()) size++;
      tables=new String[size];
      rs=dbmd.getTables(null, null, "%",
                       new String[]{"TABLE"});
      for(int i=0; rs.next(); i++)
         tables[i]=rs.getString("TABLE_NAME");
     }catch(Exception e)
     {
        e.printStackTrace();
     }
    return tables;
  }
}
```

Now, we need a form that prompts the user to select the correct url for the location of the database, the user name and password to log in.

```
<!DOCTYPE html>
<HTML>
    <HEAD>
        <TITLE>Database Login</TITLE>
    </HEAD>
    <BODY>
        <FORM ACTION="DBLoginInitialization.jsp"
METHOD="post">
            <STRONG>Select the Correct JDBC
Driver:</STRONG>
            <SELECT name="driver" size="1">
                <OPTION>com.mysql.jdbc.Driver</OPTION>
                <OPTION>jdbc.odbc.JdbcOdbcDriver</option>

<OPTION>oracle.jdbc.driver.OracleDriver</OPTION>
            </SELECT>
            <BR/>
            <STRONG>Select the Correct URL:</STRONG>
            <SELECT name="url" size="1">

<OPTION>jdbc:mysql://classdb.mads.bloomu.edu/ylu</option>
                <OPTION>jdbc:odbc:correctURL</OPTION>
            </SELECT>
            <BR/>
            <STRONG>Enter Your User Name:</STRONG>
               <INPUT type="text" name="username"
                                       value="ylu"/>
            <BR/>
            <STRONG>Enter Your Password:</STRONG>
             <INPUT type="password" name="password"
                                       value="ylu"/>
            <BR/> <INPUT type="submit" value="Login"/>
            <INPUT type="reset" value="Set back"/>
        </FORM>
    </BODY>
</HTML>
```

This form activates the DBLoginInitialization page which attempts to connect to the database supplied by the user. If the connection is successful, it forwards to the Table page; otherwise, it sends out an error massage.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<jsp:useBean id="dBBean" scope="session"
class="newpackage.DBBean">
    <jsp:setProperty name="dBBean" property="*"/>
</jsp:useBean>
```

```
<!DOCTYPE HTMLl>
<HTML>
    <HEAD>
        <TITLE>Database Login Initialization</TITLE>
    </HEAD>
    <BODY>
        <%--Make connection to the database --%>
        <% dBBean.initializeJdbc(); %>
        <% if(dBBean.getConnection()==null)
        {%>
          Error: Login failed. Try again.
        <%
        }
         else
        {%>
        <jsp:forward page="Table.jsp"/>
        <%}%>
    </BODY>
</HTML>
```

The Table page provides a list of the tables in the database and allows the user to click the submit button so that the BrowseTable page is activated. The BrowseTable page displays the table selected by the user in the previous page.

```
<!--Table.jsp this page uses the getTables() method from
the DBBean class to
    get all the table names from the database and display
them in a dropdown
    list. After the user selects a table name from the
dropdown list and clicks
    the Browse Table button, it activates the
BrowseTable.jsp
  -->

<!DOCTYPE HTML>
<%@page import="newpackage.DBBean"%>
<jsp:useBean id="dBBean" scope="session"
class="newpackage.DBBean"/>
<HTML>
  <HEAD>
     <TITLE>Select a table</TITLE>
  </HEAD>
  <BODY>
     <%String[] tables=dBBean.getTables();
       if(tables==null)
       {%>
       <STRONG> NO Table At All!</STRONG>
```

```
      <%}
        else
        {%>
         <FORM action="BrowseTable.jsp" METHD="POST">
           <STRONG>Select a table from the
list</STRONG></BR>
           <SELECT name="tablename" size="1">
             <%for(int i=0; i<tables.length; i++)
               {%>
               <OPTION><%=tables[i]%></OPTION>
             <%}%>
            </SELECT></BR></BR>
      <%}%>
           <INPUT type="submit" VALUE="Browse Table"/>
           <INPUT type="reset" VALUE="Reset"/>
         </FORM>
    </BODY>
 </HTML>


<%--
    Document   : BrowseTable
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="java.sql.*" %>
<jsp:useBean id="dBBean" scope="session"
class="newpackage.DBBean"/>
<!DOCTYPE HTML>
<HTML>
    <HEAD>
        <TITLE>Display Table</TITLE>
    </HEAD>
    <BODY BGCOLOR= "#ffff00">
      <%
          String
tablename=request.getParameter("tablename");
          ResultSet
rs=dBBean.getConnection().getMetaData().
                  getColumns(null, null, tablename, null);
      %>
      <TABLE border="2">
          <tr>
              <%while(rs.next())
               {%>
                 <td><%=rs.getString("COLUMN_NAME")%> </td>
                <%}%>
          </tr>
          <%
            Statement st
```

```
                =dBBean.getConnection().createStatement();
              rs=st.executeQuery("SELECT * FROM "+tablename);
                int n=rs.getMetaData().getColumnCount();
                while(rs.next())
                {
                    out.print("<tr>");
                    for(int i=0; i<n; i++)
                    {%>
                     <td><%=rs.getObject(i+1)%></td>
                   <%}
                     out.print("</tr>");
                }%>
        </TABLE>
      </BODY>
</HTML>
```

Review Questions

1. What is a JSP?
2. JSP scripting constructs are used in a jsp to specify java code. Describe three types of JSP scripting constructs carefully (name, syntax and what is used for).
3. What is a JSP expression?
4. What is a JSP scriptlet?
5. What is a JSP declaration?
6. What are the differences between the variables declared in a JSP declaration and the ones declared in a JSP scriptlet?
7. What is a predefined JSP variable?
8. One of the JSP predefined variable is request. What does this variable represent?
9. One of the JSP predefined variable is response. What does this variable represent?
10. One of the JSP predefined variable is out. What does this variable represent?
11. One of the JSP predefined variable is session. What does this variable represent?
12. What is a JSP directive?
13. What is a JSP page directive for?
14. What is a JSP include directive for?
15. What is a JavaBean?
16. What is a JSP action for?
17. What happens when a JSP useBean action is processed?
18. The JSP useBean action can be used in two ways. What are the different effects of these ways?
19. What is a JSP setProperty action for?
20. Describe each attribute and its possible value of the JSP setProperty action.

21. What is the JSP getProperty action for?
22. What is the JSP include action for?
23. What is the JSP forward action for?

Programming Projects

1. Design a **website**, a **database** and some **bean(s)** by using jsp's to support electronic banking. First, you need to have a database that contains the information of the customers. This database needs to be well designed with appropriate table structures that contain all customer information including customers' id, User Name, Password, name, address, the balance of each of the three accounts (Saving, Checking, and Money Market) of each customer if s/he has any, and the transaction information on each account made by each customer. Your webpage will allow the customers to open a new account, deposit to or withdraw from an account, transfer from one account to another and list all transactions made to an account within a certain time period supplied by the customer. When a customer first opens your webpage, it should ask the customer if s/he wants to login or open a new account. If a customer wants to open a new account, you should ask the customer to enter his/her name address, a user name, a pass word, what kind(s) of account(s) to open and how much to deposit into each new account. Of course, you should store all information supplied by the customer into your database. If a customer wants to deposit some money, your webpage should show a list that includes all the accounts the customer has so s/he can select an account depositing into. If a customer wants to withdraw, your webpage should show the list of accounts the customer has and the corresponding balance in each account so that your customer may select an account to withdraw from. If a customer wants to look at the transactions within a period, you should ask the customer to select an account, enter the beginning date and ending date of the period. If a customer wants to transfer money from one account to another, your webpage should show the list of the accounts with the balances that the customer has and allow the customer to select the one s/he wants to transfer from, then show the list again for the customer to select the one s/he wants to transfer to.
2. Use JSP, JavaBean and JDBC to design the WeChat program specified in the previous chapters.

Chapter Ten
Extensible Markup Language (XML)

XML, the acronym of Extensible Markup Language, is a mechanism for describing data and is independent of any programming language. XML is similar to HTML with a different purpose. XML is designed to describe data, with focus on what data is while HTML is designed to display data, with focus on how data looks. The XML format is standardized and a variety of programs can easily generate and parse XML data. In the real world, computer systems and databases contain data in incompatible formats. XML data is stored in plain text format and provides a software and hardware independent way of storing data and makes it much easier to create data documents that can be shared by different applications and to expand or upgrade to new operating systems, new applications, or new browsers without losing data. Since XML data can be read by different incompatible applications, exchanging XML data greatly reduces the complexity.

10.1 Structure of an XML Document

Like HTML, XML also consists of tags enclosed in angle brackets which go by pairs. HTML is case insensitive, but XML is case sensitive. Like HTML5, an XML start-tag must either have an end-tag or a slash character at the end of the start-tag within the angle brackets. If an attribute value of HTML does not contain spaces, it does not have to be in quote while attribute values of XML tags must be enclosed in single or double quotes.

Every XML document must start with a declaration which specifies its version:
<?xml version= “1.0”?>.

The actual data of an XML document must follow within a root element and each element has one of the two forms:
<elementName> content </elementName> or
<elementName/>.

The content of an element can be text, elements or a mixture of both:
<p>Use XML for <strong>DESCRIBING</strong> data</p>.

As having mentioned earlier, an element may have attributes: a name and a value for providing information about the element content. For example, we can write a simple XML document that describes this book:

```
<?xml version= “1.0”?>
<book>
```

```
  <title>Advanced Java</title>
  <description>Textbook</description>
  <price>$79.99</price>
</book>
```

or

```
<?xml version= “1.0”?>
<book title= “Advanced Java” description= “Textbook” price=
“$79.99”/>
```

or

```
<?xml version= “1.0”?>
<book>
  <title>Advanced Java</title>
  <description>Textbook</description>
  <price currency= “USD”>79.99</price>
</book>
```

XML tags can be nested and it is a good habit to use indentation to show the nesting structure although it is not required. Multiple tags in an XML document may share the same command or name. For example, we can design an XML document for an invoice that has an invoice number, a shipping address, a billing address and a list of items ordered.

```
<?xml version= “1.0”?>
<invoice number= “12345”>
     <address>
          <name>Youmin Lu</name>
           <street>400 East Second Street</street>
           <city>Bloomsburg</city>
           <state>Pennsylvania</state>
           <zip>17815</zip>
     </address>
     <billingAddress>
          <name>Youmin Lu</name>
          <street>Lucky Avenue</street>
          <city>Auspicious Town</city>
          <state>A New State</state>
          <zip>88888</zip>
     </billingAddress>
     <items>
           <item>
                <book>
                    <title>Big Java</title>
                    <price currency= "USD">129.99</price>
                </book>
                <quantity>30</quantity>
           </item>
            <item>
                <book>
                    <title>Data Structure using C++</title>
                    <price currency= "USD">89.99</price>
                </book>
                 <quantity>25</quantity>
           </item>
      </items>
</invoice>
```

An XML document allows multiple elements share the same name. In this document, two elements share the same name item.

Here is a summary for the differences between an XML document and an HTML document:

1. An XML document must start with the declaration statement <?xml version= "1.0"?> while an HTML5 document must start with the declaration Statement <!DOCTYPE html>
2. XML is for describing data while HTML is to structure data displayed by browsers.
3. XML is case sensitive while HTML is not.
4. Both XML and HTML consist of tags or elements.
5. An XML tag must either have an end tag or a forward slash at the end of a beginning tag. HTML5 follows this rule while the earlier HTML versions don't need every tag to have an end tag or forward slash.
6. The value of an attribute in XML must be either in double quotes or single quotes while the value of an attribute in the earlier version of HTML may not be in quotes if it does not contain spaces.

10.2 Parsing XML Document using Java Programs

A parser is a program that reads a document, checks whether it is syntactically correct or not, and takes some action as it processes the document. The Java library provides programs used to parse an XML document.

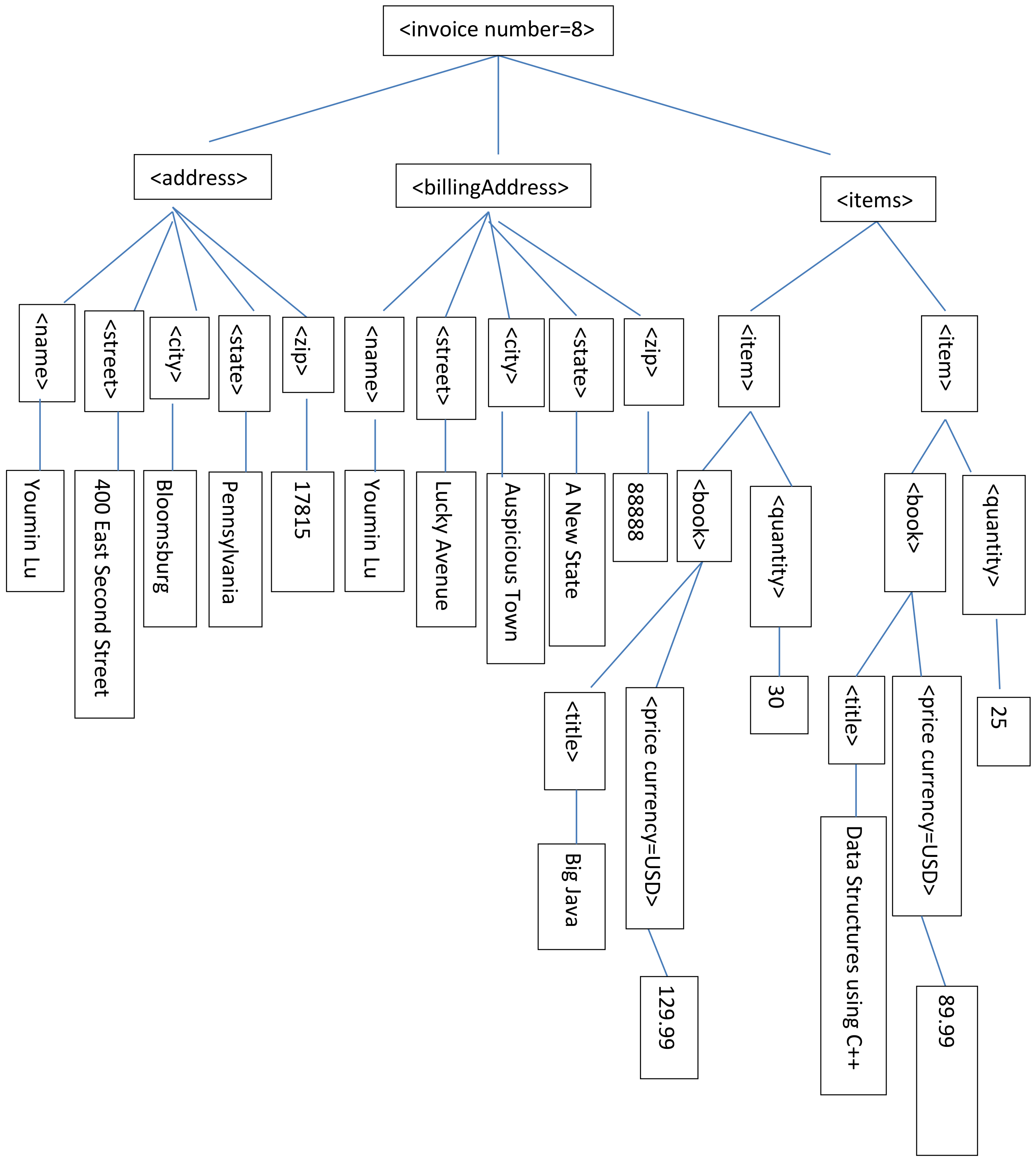
The first step to parse an XML document using Java programs is to read the XML file and store it in a Document object. To do so, you need to obtain a

DocumentBuilder (from the javax.xml.parsers package) by using the instance method newDocumentBuilder() from the DocumentBuilderFactory class (from the javax.xml.parsers package). A DocumentBulderFactory object can be obtained by using the static method newInstance of the class.

DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
DocumentBuilder builder=factory.newDocumentBuilder();

Once a DocumentBuilder object is obtained, you may use its following methods to read and store an XML file into a Document (from the org.w3c.dom package) object:
Document parse(File file);
Document parse(URL url);
Document parse(InputStream stream).

A Document stores an XML document in a tree structure with root and leaves. For example, the XML invoice document is stored in the following format.



A Document can be accessed by using methods from the XPath class (in the javax.xml.xpath package) and XPath expressions. To obtain a XPath object, we need to use the static method newInstance() of the XPathFactory class (in the javax.xml.xpath package). Then, we may use the instance method newXPath() of the XPathFactory class to get an XPath object.
XPathFactory xpfactory=XPathFactory.newInstaqnce();

```
XPath xpath=xpfactory.newXPath();
```

Once we have an XPath object, we can use its evaluate method to access and get content of the XML document:
String evaluate(String XPathExpression, Document coc)—Retursn a string that represents the content specified by the given XPathExpression from the given Document.
An XPath stores the child nodes of a node in an array starting at index 1 and the syntax of XPathExpression is similar to directory paths. For example, the XPathExpression /invoice/items/item[2]/quantity represents number 25, and /invoice/items/item[2]/book/price represents number 89.99.You may count the number of children of a node. For example, count(/invoice/items/item) gives the total number of items with name item, and count(/invoice/items/*) gives the total number of items including child nodes which are not named item. You may get an attribute value by placing @ in front of the attribute name. For example, the XPathExpression /invoice/items/itme[2]/book/price/@currency represents USD. You may also get the name of a child of a node by using an asterisk. For example, the XPathExpression name(/invoice/ items/item[1]/*[1]) represents the name of the first child of the first item.

Now, let's summarize this section by using a program and its output to show how the Java parser works.

```
import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathFactory;
import org.w3c.dom.Document;

public class XMLParser
{
   public static void main(String[] args) throws Exception
   {
     DocumentBuilderFactory
         factory=DocumentBuilderFactory.newInstance();
     DocumentBuilder builder
                =factory.newDocumentBuilder();

     Document doc=builder.parse(new File("Invoice.xml"));

     XPathFactory xpfactory=XPathFactory.newInstance();
     XPath path=xpfactory.newXPath();
System.out.print("You have ordered "+
     path.evaluate("/invoice/items/item[1]/quantity",
     doc))+“ \“"+
     path.evaluate("/invoice/items/item[1]/book/title",
     doc)+"\” books for "+
     path.evaluate("/invoice/items/item[1]/product/price",
     doc)+” “
     path.evaluate("/invoice/items/item[1]”+
                 “/product/price/@currency", doc)+
                      " dollars each.\n");
System.out.println("You have ordered "
        +path.evaluate("count(/invoice/items/item)", doc)+
                          " items.\n");
System.out.println("You have ordered "+
   path.evaluate("count(/invoice/items/*)", doc)
                     +" items including non-item.");
System.out.println("The name of the second child of the”
       +” first item is "
      +path.evaluate("name(/invoice/items/item[1]/*[2])",
       doc) +".\n");
System.out.println("The currency used is "
      +path.evaluate("/invoice/items/item[2]/book”
      +“/price/@currency", doc)+“.\n”);
    }
}
```

After this program is executed, the output should be the following.

```
You have ordered 30 Big Java books for 129.99 USD dollars each.
You have ordered 2 items.
You have ordered 2 items including non-item.
The name of the second child of the first item is quantity.
The currency used is USD.
```

Review Questions

1. What is XML?
2. What are the major differences between XML and HTML?
3. What is the general syntax of XML?
4. What is an XML parser?
5. A Java parser reads an XML file and stores it in a Document object. What is the structure of the file in the object?
6. What are the steps for using a Java parser to parse an XML document?

7. We need to use the instance method from the DocumentBuilderFactory class to get a DocumentBuilder object. How do we get a DocumentBuilderFactory object?
8. A Document is accessed by using instance methods from the XPath class. How do we get an XPath object?
9. The evaluate method from the XPath class is essential for accessing and retrieving content of the XML document. Describe this method carefully.

Programming Projects

1. Design an XML document that describes the following file about students:
   University: Bloomsburg
   College: Science and Technology
   Department: Mathematics, Computer Science and Statistics
   David Yang
   Edward Zhang
   Linda Hua
   Departement: Physics
   Mike Li
   Andrew Sun
   College: Liberal Art
   Department: Psychology
   Michel Xin
   Laura Luo
   Department: Sociology
   Lina Zhao
   Christina Zhou
   Christine Wang
2. Write a Java program that reads and parse your XML document back to the origin format.

References

1. Cay Horstmann, Big Java, John Wiley & Sons Inc.
2. Y. Daniel Liang, Introduction to Java Programming, Pearson Education, Inc.
3. Paul S. Wang, Java with Object-Oriented Programming and World Wide Web Applications, PWS Publishing
4. Walter Savitch, Java—an introduction to problem solving and programming, Pearson
5. Paul Deitel and Harvey Deitel, Java—how to program, Pearson
6. http://java.about.com
7. http://docs.oracle.com/javase/tutorial/jdbc/basics
8. Art Gittleman, Objects to Components with the Java Platform, Scott/Jones Inc.
9. Walter Savitch and Frank M. Carrano, Java—An Introduction to Problem Solving & Programming, Pearson
10. John R. Hubbard and Anita Huray, Data Structures with Java, Pearson
11. Tony Gaddis and Godfrey Muganda, Starting out with Java From Control Structures through Data Structures, Addison Wesley
12. Tony Gaddis, Starting with Java Early Objects, Addison Wesley
13. John Lewis, Peter DePasquale and Joseph Chase, Java Foundations—Introduction to Program Design and Data Structures, Pearson
14. Lewis Loftus, Java Software Solutions—Foundations of Program Design, Addison Wesley
15. David Barnes and Michael Kolling, Objects First with Java, Pearson
16. Joyce Farrel, Java Programming, Course Technology
17. Stuart Reges and Marty Stepp, Building Java Programs—A Back to Basics Approach, Pearson
18. Cay Horstmann, Big Java, John Wiley & Sons Inc.
19. Y. Daniel Liang, Introduction to Java Programming, Pearson Education, Inc.
20. Paul S. Wang, Java with Object-Oriented Programming and World Wide Web Applications, PWS Publishing
21. Walter Savitch, Java—an introduction to problem solving and programming, Pearson
22. Paul Deitel and Harvey Deitel, Java—how to program, Pearson
23. http://java.about.com
24. http://docs.oracle.com/javase/tutorial/jdbc/basics
25. Art Gittleman, Objects to Components with the Java Platform, Scott/Jones Inc.
26. Walter Savitch and Frank M. Carrano, Java—An Introduction to Problem Solving & Programming, Pearson
27. John R. Hubbard and Anita Huray, Data Structures with Java, Pearson

28. Tony Gaddis and Godfrey Muganda, Starting out with Java From Control Structures through Data Structures, Addison Wesley
29. Tony Gaddis, Starting with Java Early Objects, Addison Wesley
30. John Lewis, Peter DePasquale and Joseph Chase, Java Foundations—Introduction to Program Design and Data Structures, Pearson
31. Lewis Loftus, Java Software Solutions—Foundations of Program Design, Addison Wesley
32. David Barnes and Michael Kolling, Objects First with Java, Pearson
33. Joyce Farrel, Java Programming, Course Technology
34. Stuart Reges and Marty Stepp, Building Java Programs—A Back to Basics Approach, Pearson