COMP 132 study guide

Topic 1a: Objects and Classes

In this course, we study the **Java** programming language.

We will use Java within a program called Eclipse. Eclipse is an **integrated development environment** (IDE), meaning that it can be used to write, test and run computer programs.

Java is an **object-oriented** programming language, which means that its programs use objects.

An **object** is a collection of **data**, and a list of **actions** that can be performed on the data. Every object belongs to a class. A **class** is a description of all possible objects with the same type of data and the same list of actions, together with some recipes (called **constructors**) for creating new objects. An object that belongs to a class is called an **instance** of that class. Two different instances of the same class can have different values for their data (e.g. color = “green” vs color = “red”), but they must have the same type of data and the same list of actions.

Informally speaking, a class is a description of a category of objects, whereas an object is a representation of a specific item from the class.

In Java, the actions that can be performed on an object's data are called **methods**. To perform an action on an object, you **invoke** or **call** the relevant method.

Most methods need to be given extra information describing the action they must perform. For example, a method that deposits money into a bank account must be told the amount of the deposit. Each piece of information that must be given to a method is called a **parameter**. Each parameter has two parts, (i) a **data type** describing what sort of information it stores, and (ii) a **parameter name**, which is the name Java will use to refer to the parameter.

In Java, each method lists all the parameters it needs in parentheses after the method name. This list of parameters is called the **parameter list**. Common data types for parameters in Java include String and int. (String is used for storing strings of characters, like “abcd” or “green”; int is used for storing whole numbers like 8 or -53. In Java, strings must be surrounded by quotation marks.) Here are two examples of parameter lists: The parameter list for a deposit method might be (int amount). The parameter list for a changeColorAndSize method might be (String newColor, int newSize).

Topic 1b: Main Method

In order to run a Java program, one of your classes must have a **main method**, which is a special method that is automatically invoked when a Java program is run. The main method always has the following signature:

 public static void main(String[] args)

We study public, static, and “String[] args” later in the course.

The code in the body of the main method is executed when the program starts. Typically, the body of the main method first creates a new object and then executes one or more of that object’s methods. The following example code uses several concepts that have not yet been studied, but it is useful as an initial explanatory example. The lines beginning with “//” our comments, that explain subsequent lines of source code. Here then, is an example of a possible main method of an Account class:

 public static void main(String[] args) {

 // Create a new Account object with a balance of $50 (5000 cents)

// and interest rate of 12%

 Account account = new Account(5000, 12);

 // Withdraw two dollars from the account

 account.withdraw(200);

 // Get the new balance and store it in a local variable

 int newBalance = account.getBalance();

 // Print out the new balance

 System.out.println("New balance is " + newBalance);

 }

To **run a program** with a main method in Eclipse, select Run | Run As | Java Application (or simply Run | Run on subsequent runs).

Topic 1c: Object state

Recall that an object consists of some data and a list of actions that can be performed on the data. An object's data is also called its “state”, “attributes”, “instance variables” or “fields”. All of these words mean basically the same thing, but “fields” is the most commonly used technical term in computer programming. Each **field** in an object stores exactly one piece of data. Just like a parameter, a field has a data type (e.g. String or int) describing what sort of information it stores, and a name.

Recall that a class is a description of all objects with the same type of data and the same list of actions, together with some recipes (called **constructors**) for creating new objects. In Java, you write a class description in a file whose name ends in “.java”. The file has three main components:

* Fields
* Constructors
* Methods

The content of a .java file is called Java **source code**, or just **code**. You can view the source code for a class in Eclipse by double-clicking on it in Package Explorer.

Topic 1d: Primitive and object datatypes

As we know, every field and every parameter have a data type. In Java, there are two kinds of data types:

* **primitive data types**, which are built in to Java (e.g. int)
* **object data types**, which are defined by programmers (e.g. Circle)

Every class defines its own new object data type. For example, the file Circle.java defines the Circle data type. Parameters and fields can have object data types. For example, a Picture object might have a field “Circle sun”—a field whose name is sun and whose data type is Circle.

Topic 1e: Accessors and mutators

Recall that every method performs an action on the data of an object. Some methods *change* the data—these are called **mutator** methods. Other methods merely *examine* the data and return some useful information about it—these are called **accessor** methods. The type of data returned by a method is called its **return type**. The value returned by a method is called its **return value**. For example, the getBalance() method of a BankAccount class might have a return type int, and invoking getBalance() on some particular bank account might result in the return value 57. Methods that don't return any information have the special return type **void**. Therefore, mutator methods usually have return type void. Accessor methods usually have a non-void return type.

So parameters describe the *inputs* to a method, and the return type describes the *output* of a method. The inputs and outputs of a method are summarized by the **method signature**, which consists of

* return type
* method name
* parameter list

The precise format of a method signature is as follows:

 returnType methodName ( parameterList )

For example, in the method signature

 void enrollStudent ( Student newStudent )

the return type is void, the method name is enrollStudent, and the parameter list consists of a single parameter whose name is newStudent and whose data type is Student.

You can view a summary of the method signatures (and certain other information) of a class in Eclipse via the Outline view (Window | Show View | Outline).

Topic 1f: Fields and Assignment Statements

Methods can provide information about an object in two very different ways: (i) as a return value, and (ii) by printing information on the screen. Information printed on the screen can only be used by humans who read it, whereas information provided as a return value can be used by other parts of a Java program.

To write Java programs, you need to know the correct names of certain punctuation characters:

* ( ) – parentheses
* { } – **braces** (or “curly braces”)
* [ ] – **brackets** (or “square brackets”)
* ; —semicolon
* / —forward slash

The source code for a class definition in Java has the following structure (note the braces):

 public class ClassName {

 field definitions

 constructor definitions

 method definitions

}

A field definition in Java has the following structure (note the semicolon):

 private dataType fieldName;

For example,

 private int numStudents;

Any line of Java code that starts with // (two forward slashes) is called a **comment**. Comments are ignored by the compiler but are useful for humans reading the code. If // occurs in the middle of a line, then everything after the // is also a comment.

Any code that occurs between the two special symbols /\*\* and \*/ is also treated as a comment by the compiler, but it is a special type of comment called a **Javadoc comment**. Javadoc comments should be provided for every class and every method. Javadoc comments are used to automatically create helpful documentation for human readers.

To store a value in Java, you use an **assignment statement**. The structure of an assignment statement is:

 locationName = value;

This might look backwards until you get used to it: note that the value on the *right* gets stored in the location named on the *left*. The only locations for storing values that we know about right now are fields. For example,

 numStudents = 5;

This stores the value 5 in the field called numStudents.

The = (equals sign) in an assignment statement is called the **assignment operator**. When reading Java code out loud, the assignment operator should always be read as “gets the value” (not “equals”), because an assignment statement moves data from one place to another. For example, the Java code numStudents = 5 should be read out loud as “numStudents gets the value five.”

The value that appears on the right-hand side of an assignment statement is known as an **expression**. An expression can be quite complicated. For example, in the assignment statement

numStudents = 5 + numJuniors - numSeniors;

the code 5 + numJuniors – numSeniors is an expression.

The two sides of an assignment statement must have the same data type. For example, you cannot assign a value of type String to a field of type int.

Whenever a method or constructor with parameters is invoked, the parameters are given some particular values. Those particular values are called **actual parameters** or **arguments**. When the source code of a method is considered on its own, the parameters do not have any particular values, and are referred to as **formal parameters**. For example, a method whose signature is

 void deposit ( int amountDeposited )

has one formal parameter (amountDeposited). Whenever the method is invoked, amountDeposited will be given an actual value (e.g. 535), which is the actual parameter or argument.

In Java code, the names of formal parameters and fields can only be used in certain zones. The zone in which a name can be used is called its **scope**. The scope of a formal parameter is its method or constructor definition. The scope of a field is its class definition.

In Java, a **statement** is a piece of code that ends in a semicolon (and doesn't contain other semicolons). A **block** of code is a sequence of statements enclosed in curly braces.

Topic 1g: Defining and invoking constructors

The structure of a constructor definition in Java is:

Javadoc comment

public ClassName ( parameterList )

{

constructorBody

}

The constructor body is a block of code that is executed whenever the constructor is invoked. The main purpose of the constructor body is to **initialize** the fields, which means they are given definite initial values. If the initial value of a field is always the same, it is called a **default** value (e.g. maxStudents = 5 gives the field maxStudents the default value 5). Otherwise, the initial value of a field is usually assigned the value of one of the constructor's actual parameters (e.g. if a constructor has a formal parameter named initMaxStudents, the constructor body might include the statement maxStudents = initMaxStudents).

To call a constructor in Java, use the following pattern:

new ClassName ( parameters );

The Java keyword **new** is known as the "new operator". It is always followed by a constructor and creates a new object. Usually, the new object is assigned to a reference so it can be used later. For example, if the Account class has a constructor with signature Account(int balance, int interestRate) then the following code

Account savings;

savings = new Account(5000, 12);

creates a new Account object with balance 5000 and interest rate 12 and stores a reference to the new object in the local variable savings.

Topic 1h: Defining Methods

The structure of a method definition in Java is:

Javadoc comment

public returnType methodName ( parameterList )

{

methodBody

}

The method body is a block of code that is executed whenever the method is invoked.

Accessor methods return a value by using Java's **return** statement. For example, the statement

return 34;

returns the value 34. The statement

return numStudents;

returns the value stored in the field numStudents.

To ensure code is easy to read, each block must be **indented** from the enclosing block of code. Eclipse will often guess the correct indentation while you are writing code. You can also format your entire Java file with correct indentation by selecting Source | Format within Eclipse.

Whenever the compiler discovers an error in your code, it issues an error message, and Eclipse highlights the location where the compiler got confused. The error message shown in the Problems view is sometimes helpful (e.g. “missing semicolon”), but often it is misleading for inexperienced programmers (e.g. “illegal start of expression”). Clicking on the lightbulb at the start of the line containing the error shows a list of suggested fixes, which can be useful.

Topic 1i: Printing, Strings and Integer Division

In Eclipse, the output of a program is displayed in the Console view. Java programs display output by using the built-in System.out.println method.

In Java, two strings can be joined together, or **concatenated**, using a + (plus sign). When used between two strings, a plus sign is called the **concatenation operator**. A plus sign also functions as a concatenation operator when used between a string and a number.

But when a plus sign is used between two numbers, it functions as an **addition operator**, and produces the sum of the two numbers. Arithmetic in Java is performed using the following symbols:

* addition : +
* subtraction: -
* multiplication: \*
* division: /
* remainder: %

You can use parentheses to prioritize arithmetic. For example, 5 \* (2 + 4) evaluates to 30, but (5 \* 2) + 4 evaluates to 14.

Data types have an extremely important effect on arithmetic performed in Java. In particular, *dividing one integer by another integer always produces an integer*. This is called **integer division**. For example, 19 / 5 evaluates to 3, because you can fit three 5’s into 19 and have 4 left over. The amount left over is called the remainder, and in Java is produced by the **remainder operator** (%). So in Java, 19 % 5 evaluates to 4.

Topic 1j: double and default constructors

**double** is a primitive data type for storing numbers that aren't necessarily integers, like 8.32 or -432.1.

**boolean** is a primitive data type for storing one of the two possible values **true** and **false**.

A **default constructor** is a constructor that has no parameters, and therefore must assign a default value to every field.

Constructors that are not default constructors have at least one parameter, and the parameters are used to set the initial values of some of the fields. Other fields might still be given default values. Non-default constructors are sometimes referred to by the number of parameters, or arguments, they have. For example, a constructor with 4 parameters is called a **4-argument constructor**.

Constructor parameters are often named using the following convention: take the field name, capitalize it, and add the string “init” in front. For example, if there is a field called numWombats and then the constructor parameter specifying the initial value for this field would typically be called initNumWombats. The letters “init” remind us that this parameter specifies the *initial* value of the field.

Topic 2a: Unit Testing with JUnit

Compilers can only detect elementary errors in your code. Other errors (also called **bugs**) must be found by **testing**, which means running the code and checking that it does what you expect. Testing manually is very tedious, so software developers use automated testing. **Junit** is a popular automated testing tool for Java, and Eclipse provides convenient ways of using it.

To perform JUnit testing in Eclipse, you first create a **test class** (also known as a **test case**), and then create a **test method** for every automated test to be run. There should be a separate test method for every constructor and every mutator method. Some mutator methods require more than one test method. Accessor methods do not require test methods.

The most important rule of software development is:

code a little, test a little

In other words, *never start writing a new constructor or a new method until you have fully tested the one you just wrote*. In order to test a constructor, you must first write all the accessor methods: this is the only exception to writing new code before testing existing code.

The main reason for “code a little, test a little” is to save you time: bugs in existing code tend to cause bugs in new code, so by eliminating bugs as early as possible you reduce the total number of bugs.

For each kind of Java code that we learn about, there will be a **testing rule** that states how to test that kind of Java. Recall that testing requires you to check that the code did what you expected; this check is called an **assertion**. For example, you can assert that the return value of the getNumWombats() method is 5.

The testing rule for constructors and accessors is:

* one test method for each constructor
* create an object using the constructor being tested
* call the accessor for each field and make an assertion

Topic 2b: Multiple tests and local variables

Some mutator methods have several different kinds of behavior that are possible. To test a method that has multiple behaviors, you need a separate test method for each behavior.

A **local variable** is a named location where you can temporarily store a value. You must **declare** a local variable before you can use it for storing values. The format for a local variable declaration is:

dataType variableName;

For example,

 int newAngle;

is a local variable declaration. A local variable must be declared *inside* a method. The scope of a local variable is the method in which you declare it. So you cannot refer to a local variable outside its own method. Also, you must declare a local variable before you refer to it.

It is possible to store an initial value in a local variable at the same time as declaring it. The format for simultaneously declaring and initializing a local variable is:

dataType variableName = initialValue ;

For example,

 int newAngle = 85;

 int anotherAngle = angle + degrees;

are two valid local variable declarations

Topic 3a: Conditional statements and statement coverage

In Java, a **Boolean expression** is a piece of code that evaluates to either true or false. For example, balance – amount > 0 is a Boolean expression.

Boolean expressions often use **relational operators** to compare values. The relational operators in Java are:

* < less than
* <= less than or equal to
* > greater than
* >= greater than or equal to
* == is equal to
* != is not equal to

Note the difference between “==” and “=”. The double equals-sign “==” tests whether two things are equal. A single equals-sign “=” stores a value in an assignment statement. Using “=” when you intend “==” is a common programming error.

In Java, the format of a **conditional statement**, also known as an “**if statement**”, is:

if (condition) {

 statements to execute if condition is true

}

else {

 statements to execute if condition is false

}

In the above code, the condition in parentheses must be a Boolean expression. Each block of code between braces is called a **clause**. So a conditional statement has an “if clause” and an “else clause”. The else clause is in fact optional and can be omitted, resulting in a conditional statement as follows:

if (condition) {

 statements to execute if condition is true

}

An important goal of software testing is to achieve **statement coverage**. This means that your suite of tests causes every line of code to be executed at least once. This principle leads to the **testing rule for conditional statements**:

* tests must execute every clause in a conditional statement

Topic 3b: Nested and Cascading Conditional Statements

To test for more than two possibilities, you can use a **cascading if statement**:

if (condition1) {

 statements to execute if condition1 is true

}

else if (condition2) {

 statements to execute if condition2 is true

}

.

.

.

else if (conditionX) {

 statements to execute if conditionX is true

}

else {

 statements to execute if all conditions are false

}

Another way to test for multiple possibilities is to insert an if statement inside one or more of the clauses of another if statement. This is called a **nested if statement**. For example, the following format uses a nested if statement to test for 4 different possibilities:

if (condition1) {

 if (condition2) {

statements to execute if condition1 is true and condition2 is true

}

else {

statements to execute if condition1 is true and condition2 is false

}

}

else {

 if (condition3) {

statements to execute if condition1 is false and condition3 is true

}

else {

statements to execute if condition1 is false and condition3 is false

}

}

Topic 3c: Boolean Operators

Boolean expressions can be combined using **Boolean operators**:

* && and
* || or
* ! not

The technical meanings of these operators are:

|  |  |
| --- | --- |
| P && Q | true only if P is true and Q is true |
| P || Q | true if P is true, or if Q is true, or if both are true |
| !P | true if P is false |

Here are some examples of Boolean expressions that use Boolean operators:

* (x > 3) && (x < 10)
* (x == 2) || (x == 3)
* !(x == 2)
* (x != 2) || !(x < 3)

Topic 3d: Debugging

A **debugger** is a piece of software that lets you examine computer programs while they are executing. Debuggers are very useful for finding bugs in programs. Eclipse includes a debugger, which can be launched by selecting Run | Debug. Debuggers allow you to set a **breakpoint** at any line of code in a program. The program will temporarily halt at each breakpoint, giving you a chance to examine its state. Debuggers also allow you to **single step** through a program, which means the program stops after every statement. A key concept in debugging is the **call stack**. When the program stops at a breakpoint, the call stack consists of the nested structure of methods that are currently being executed. For example, if the call stack consists of main(), a(), and b(), this means that the main() method invoked method a(), which invoked method b(), which encountered a breakpoint.

Topic 3e: Composition

**Composition** is the use of objects as fields in other objects. Composition helps programmers manage complexity by allowing complex objects to be built out of simpler ones. In particular, composition is used in the following four important principles of software design:

* **Divide and Conquer:** complex problems are broken down into smaller, simpler sub-problems.
	+ benefit: it is easier to solve smaller problems.
* **Modularization**: problems are divided so that the simpler sub-problems (also called **modules**) are relatively independent, and interact in well-defined ways.
	+ benefits:
		- division of labor: independent problems can be assigned to separate people or teams
		- localized changes: a change in the way one module works often has little or no effect on other modules
* **Reuse**: existing components can be reused in new software
	+ benefit: it is less work to reuse existing code than to write new code
* **Abstraction**: a component can be used without understanding its implementation details
	+ benefit: a complex problem can be solved without any one person mastering all the details of all sub-problems

In Java, we also use **packages** to achieve modularization. A class is declared to be part of a package in the first line of the file, using the package keyword:

package animals;

public class Tiger

{...

The .java files for a given package must be stored in a directory of the same name. So in the above example, Tiger.java would be in the animals directory. Packages can also be nested, and the directory structure must mirror the package structure. For example, source code in the animals.mammals package would be placed in the mammals subdirectory of the animals directory.

Code in a particular package automatically has access to other public classes in . To access code in another package, use the package name as a prefix (e.g. new animals.mammals.Gorilla()

or import the desired classes near the top of the source file (e.g. import animals.mammals.Gorilla).

Topic 3f: Design notation and implementing composition

Recall that there are two kinds of data types in Java: primitive data types and object data types. Whenever a field, local variable, or parameter has an object data type, it does not in fact store the entire object. The object itself is stored in some other location in the computer's memory, and the field, local variable, or parameter stores only a **reference** to the object. A reference states where the object can be found in the computer's memory. You can think of it as an arrow pointing from the field, local variable, or parameter to the actual location of the object.

If a reference does not point to any object, it has the special value **null**.

An **object diagram** shows the state of some objects at a particular instant when running a computer program. Each object is represented by a box. At the top of the box, the name and data type of the object are displayed as follows:

[name] : datatype

The name is in brackets because it is optional. Sometimes an object does not have a name, and sometimes it has more than one name. So we include object names on object diagrams when they are helpful, and leave them out otherwise. Below the name and data type of the object, each field is listed as follows:

 fieldName : value

For fields that store primitive data types, the actual value is listed. For fields that store object data types, the field's reference is depicted with an arrow pointing to another object on the object diagram. Exception: Because the object data type String is used so often, we treat it as primitive when drawing object diagrams. An online example of an [object diagram](http://users.dickinson.edu/~jmac/comp132/object-diagram.jpg) is available.

A **class diagram** shows the relationship between the source code of different classes. Each class is shown as a box with the class name inside it. If the source code of one class mentions another class (for example, uses it as a field, local variable or parameter data type), then an arrow is drawn from the first class to the second class. This arrow is called a “**uses**” arrow because the first class uses the second class. An online example of a [class diagram](http://users.dickinson.edu/~jmac/comp132/class-diagram.jpg) is available.

A **static** property of a computer program depends only on the source code and not the way the program is run. A class diagram depicts a static view of a program.

A **dynamic** property of a computer program depends on the way it is run. An object diagram depicts a dynamic view of a program.

Topic 3g: Method Calls and Test Fixtures

You call a method in Java using the following **method call** syntax:

 objectReference . methodName ( actualParameters ) ;

For example, if savings is a reference to an object of type Account, which has a method with the signature void deposit(int amount), then

 savings.deposit(25);

calls the deposit method on the savings object with amount equal to 25.

The object to the left of the period in a method call is the **calling object**. It is the object on which the method operates.

A **test fixture** is a collection of objects that are automatically re-created before each test method is run. Using a test fixture saves work by automating the setup for every test. In Junit, you create a test fixture by creating a setUp() method in the test class, annotated with @Before. Eclipse can create this method automatically.

There is no automatic way to test code that prints to the console, for example using System.out.println(). To test such code, write a test that achieves statement coverage without using any assert statements. When the test executes, examine the console window and manually check that the output is as expected.

Topic 3h: this and null

As we know, every constructor or method is executed on behalf of some particular object, known as the *calling object*. In Java, the special keyword **this** can be used as a reference to the calling object. For example, this.age is the age field of the calling object, and this.increaseWeight(23) invokes the increaseWeight method on the calling object. The keyword this can often be omitted, since the compiler assumes you are referring to the calling object except in certain exceptional situations. So increaseWeight(23) almost always means the same thing as this.increaseWeight(23).

One very important use of this is to remove ambiguity when a parameter and a field have the same name. In this case, the compiler assumes you are referring to the parameter unless you use this. Programmers often use this technique for writing constructors: instead of using constructor parameters that start with “init”, you can use parameters that have the same names as the fields, and instead employ this when necessary. For example,

public Rectangle (int length, int width) {

 this.length = length;

 this.width = width;

}

A frequent programming error is to invoke a method on an object reference that is null. In other words, the calling object is null and there is no object on which the method can be performed. This results in a **NullPointerException**.

Topic 3i: public and private

A **public** field or method is visible to code in other classes.

A **private** field or method is not visible to code in other classes.

The Java keywords public and private are called **access modifiers**, because they modify the type of access that other code has to a given field or method.

The general rule for the public and private access modifiers is:

* all *fields* are declared private
* methods for use *outside* the current class are declared public
* methods for use only *inside* the current class are declared private

The principle behind this rule is an idea called **information hiding**, which states that a class should reveal to other classes only the information needed by other classes—all other information should be *hidden*, by declaring it private.

One advantage of information hiding is that it makes abstraction work better: if the implementation details of one class are hidden from another class, the second class can be written without depending on those implementation details.

The **public interface** of a class consists of its public methods and constructors—in other words, the things that are available to other classes. A public interface includes names, parameter lists, return types and specified behavior of each method and constructor. Javadoc comments before each method are the conventional way of communicating a public interface to other programmers. These Javadoc comments can be automatically extracted and formatted to present a coherent, easily understood view of the public interface, generally as a web page.

Topic 3j: Refactoring

Making small changes to some code in order to improve it or fix errors is called **maintaining** the code.

**Repeated code** is two or more sections of code that do the same thing. Sections of repeated code are not always identical—they merely achieve the same result. Repeated code is a poor programming practice because it is difficult to maintain. Specifically:

* It is a common mistake to change one instance of a repeated section while forgetting to make the same change in another repeated section.
* Even if you remember to make all the required changes, it requires additional work to do so.

To eliminate repeated code, you use a technique called **factoring out repeated code**. This consists of the following steps:

* Define a new method that performs the same action as the repeated code.
* Copy one instance of the repeated code into the new method, and make any necessary adjustments.
* Replace every instance of the repeated code with a single line that calls the new method.

**Refactoring** a class means improving the implementation of the class without changing its public interface. Factoring out repeated code is one example of refactoring, but there many others, including adding new fields or changing the data type of fields (e.g. from array to ArrayList).

Topic 4a: Collections and ArrayList

Java provides many classes for storing collections of objects; one of them is called **ArrayList**. An ArrayList stores a list of objects of a *single type*. The objects are stored in an ordered list and are numbered starting from 0. Each object in the list is called an **element**, and an element's numbered position in the list is called its **index**.

A constructor for ArrayList specifies what type of objects will be in the list, using the format:

ArrayList < ClassName > ( )

This constructor creates an empty list, where *ClassName* is the type of objects to be stored in the list.

Four important methods of the ArrayList class are:

* add( *objectReference* )— adds the given object to the end of the list
* size( )— returns the number of objects in the list
* get( index )— returns a reference to the element at the specified index
* remove( index )— removes the element at the specified index; elements after the removed element are shifted down so their indexes are reduced by one.

When calling get or remove, the specified index must be valid: it must be ≥ 0 and < the size of the list. An invalid index causes an **IndexOutOfBoundsException**.

The ArrayList class is provided with Java but is not automatically available. ArrayList is part of the **Java Class Library**. The Java Class Library is organized into **packages**. ArrayList is in the java.util package. You can make ArrayList available using the Java keyword **import**. So any Java file that uses ArrayList must first include the following line of code:

import java.util.ArrayList;

Topic 4b: Returning Objects and Testing Collections

When a method's return type is an object data type, and the method encounters an error, it is conventional to return the value null.

The testing rule for methods that add to a collection is:

* add several elements
* check the size of the collection
* check the first and last elements
* check just enough fields to verify the expected elements are present

The testing rule for methods that get by index is:

* add several elements
* test getting from a negative index
* test getting from an index that is too large
* test getting from a valid index

The testing rule for methods that remove from a collection is:

* add several elements
* remove an element from the middle of the list
* check the size of the collection
* check the element that is now at the removed index
* check just enough fields to verify the expected element is present
* test removing from a negative index
* check only the size
* test removing from an index that is too large
* check only the size

Topic 4c: Iteration with for Loops

The **increment operator** in Java is ++. It increases the value of a variable by 1. So day++ has the same effect as day = day + 1.

The **decrement operator** in Java is --. It decreases the value of a variable by 1. So day-- has the same effect as day = day - 1.

To repeatedly execute the same piece of code several times, you use a programming construct called a **loop**. Each execution of the loop code is called an **iteration**. Java provides at least three ways to program a loop: for loops, while loops, and foreach loops. We study for loops first.

The syntax of a **for loop** is:

for (initializer; loopCondition; update) {

 loopBody

}

where:

* *initializer* is executed once, before the first iteration of the loop
	+ typically initializes a **loop variable**, which keeps track of how many iterations have been performed
	+ e.g. int day = 0
* *loopCondition* is checked before each iteration of the loop, and the loop is terminated if the loop condition is false
	+ typically checks whether the loop variable is still in the expected range
	+ e.g. day < prices.size()
* *update* is executed after each iteration of the loop
	+ typically updates the loop variable ready for the next iteration
	+ e.g. day++
* *loopBody* is executed on every iteration of the loop, and can consist of many statements

For example,

for (int num = 0; num < 5; num++) {

 System.out.println(num);

}

prints the numbers 0, 1, 2, 3, 4 on separate lines.

Note carefully the execution order of the statements in a for loop. Let's use the abbreviations INIT for initializer, LC for loop condition, U for update and LB for loop body. Then the order of execution is:

INIT, LC, LB, U, LC, LB, U, LC, LB, U, LC, ... [stops the first time LC is false]

Topic 4d: Aggregating and searching collections

To compute aggregate data, use a local variable to accumulate data from every execution of a loop. For example, to add the numbers from 10 to 20 and print the result:

int total = 0;

for (int num = 10; num <= 20; num++) {

 total = total + num;

}

System.out.println(total);

The testing rule for methods that compute an aggregate result over a collection is:

* add several elements to the collection, then call the "aggregate" method and check that the result is correct
* use an empty collection to call the "aggregate" method and check that the result is correct

To modify some or all of the elements in a collection, simply iterate through the collection and perform the desired action on each element. For example,

 /\*\*

\* Increase the number of rental nights on each rented

\* DVD by 1 night.

\*/

public void increaseRentalNights() {

 // For each DVD in the store.

 for (int i=0; i<dvdList.size(); i++) {

 DVD aDVD = dvdList.get(i);

 // if it is checked out.

 if (aDVD.getNightsRented() > 0) {

 // increase its rental nights.

 aDVD.addRentalNight();

 }

 }

}

The testing rule for methods that modify elements of a collection is:

* modify the elements (call the method)
* check at least one modified element
* check at least one unmodified element (if any)
* in each case, check the field(s) that could have been modified

To **search a collection** for an object with a particular property (such as the largest value of a given field), iterate through the entire collection using one or more local variables to remember the information you have gained so far. For example,

public DVD getLongestRental() {

 // Local variable to remember the longest rental so far.

 DVD longestDVD = null;

 int longestNights = -1;

 // Iterate through the rest of the DVDs.

 for (int i=0; i<dvdList.size(); i++) {

 DVD aDVD = dvdList.get(i);

 // If the current DVD has been rented longer than

 // the longest one seen so far then set it

 // as the longest so far.

 if (aDVD.getNightsRented() > longestNights) {

 longestDVD = aDVD;

 longestNights = aDVD.getNightsRented();

 }

}

return longestDVD;

}

The testing rule for methods that search a collection is

* use one test method to test searching an empty collection
* use another test method to test searching a collection that does not contain the searched-for object
* use a third test method to test searching a collection that does contain the searched-for object

Topic 4e: While Loops

The syntax of a **while** **loop** is:

while (loopCondition) {

 loopBody

}

where:

* *loopCondition* is checked before each iteration of the loop, and the loop is terminated if the loop condition is false
	+ e.g. day < prices.size()
* *loopBody* is executed on every iteration of the loop, and can consist of many statements

For example, the code

int x = 0;

 while (x < 5) {

 System.out.println(x);

 x = x + 1;

}

prints the numbers 0, 1, 2, 3, 4 on separate lines.

A common programming error is to write a loop that never terminates. This is called an **infinite loop**. For example,

int x = 0;

 while (x < 5) {

 System.out.println(x);

 x = x - 1;

}

is an infinite loop.

Topic 4f: .equals() and Returning a Collection

Recall that the == operator tests whether two things are equal. In fact, == behaves completely differently for primitive data types and object data types. For primitive data types, == is true whenever the compared values are equal. But for object data types, == is true only when the compared references point to *exactly the same object*. If the references point to two different objects that happen to have the same values for their fields, then == is false. Programmers usually define a **.equals()** method for testing whether two objects have the same values in their fields. This is especially important for Strings. It is a common programming error to think that if we declare

String string1 = "abc";

String string2 = "abc";

then string1==string2 is true. In fact, string1==string2 could be false (because the two strings are references to different objects), but string1.equals(string2) is true (because the two strings contain the same characters).

To write a .equals() method, you need to compare the calling object (using this) with another object passed in as a parameter, as in the following example:

public class Account {

 private int balance;

 private int interestRate;

 public boolean equals(Account account) {

 if (this.balance == account.balance

 && this.interestRate == account.interestRate) {

 return true;

 } else {

 return false;

 }

 }

}

Note that Eclipse can automatically generate .equals() methods.

To return a collection of objects: (1) create a new collection, (2) add the desired objects using a loop, and (3) return the new collection. For example,

public VideoStore getAllDVDsWithRating(String rating) {

// 1. Create a new empty VideoStore

VideoStore withRating = new VideoStore(rating +

" Rated DVDs");

// 2. Iterate through the DVDs, adding the suitable ones

for (int i = 0; i < dvdList.size(); i++) {

 DVD aDVD = dvdList.get(i);

 // If the DVD has the specified rating

// add it to the new collection.

String curRating = aDVD.getRating();

if (curRating.equals(rating)) {

withRating.addDVD(aDVD);

}

}

// 3. Return the new collection.

return withRating;

}

The testing rule for methods that return collections is:

* use one test method to check that the empty collection is returned correctly
* in another test method, check that a non-empty collection is returned correctly:
* check the size of the collection
* check the first and last elements (check enough fields to be sure they are the ones you expect)

Topic 4g: Object diagrams for ArrayLists

When making an object diagram for an ArrayList, use the same techniques as for simpler object diagrams, taking care to show all references for elements in the list.

Topic 4h: foreach loops

A **foreach loop** can be used in Java to iterate over each element in any collection, using the following syntax:

for ( ElementType variableName : collectionName ) {

 loopBody

}

where

*ElementType* is the data type of elements in the collection.

*variableName* is any local variable name.

*collectionName* is the name of the collection to be iterated over.

*loopBody* is a sequence of statements that are executed once for every element in the collection. At each iteration, *variableName* is assigned to a different element of the collection automatically.

For example, assuming that ArrayList<String> nameList has been declared, the following code iterates over the names in the list, printing out each name on a separate line:

 for (String name : nameList) {

 System.out.println(name);

 }

Topic 4i: Wrapper types for primitive types

Java collections such as ArrayList can store collections of object types, but not primitive types. Because of this, Java provides an object **wrapper** class for each primitive type. For example, Integer is the wrapper class for int, and Double is the wrapper class for double. Thus, we can store integers in an ArrayList by creating an ArrayList<Integer>. In many cases, the Java compiler will automatically convert between a primitive type and its corresponding wrapper; this is called **autoboxing**.

Topic 5a: Class constants

It is often useful to attach a name like MAX\_TEMPERATURE, NUM\_STUDENTS, or BUILDING\_NAME to a particular value such as 71.5, 64454, or “Tome”. A named value is called a **class constant** or just a **constant**. Class constants are declared using the two keywords **static final**, as follows:

public static final DataType NAME = value;

For example,

public static final double MAX\_TEMPERATURE = 71.5;

public static final String BUILDING\_NAME = "Tome";

By convention, every letter of a constant’s name is capitalized, with underscores separating words.

The **static** keyword means that all instances of the class share one copy of the field (in contrast to non-static fields, which have a separate copy for every instance of the class).

The **final** keyword means that the value of the field cannot be changed after it is initialized.

The combination static final therefore means that a single copy of the field is shared between all instances of the class and its value cannot be changed.

Inside the class where a constant is defined, you refer to a constant by its name just like any other field. Outside the class, you refer to a constant using the syntax ClassName.ConstantName.

The two main reasons for using class constants are:

* the code is more easily understood (e.g. MAX\_TEMPERATURE is more meaningful to a human reader than 71.5)
* it is much easier to change the value of a constant (which occurs at only one place in the code) than it is to change a value in every place that it is used in the code

Therefore, you should avoid using literal values in your code. (A **literal value** is a number or quoted string, like 23, 71.5 or "Tome".) Instead, define a class constant that equals the literal value.

Topic 5b: Static fields and methods

If a field is declared **static**, there is only one copy of this field for the class. All instances of the class share a single copy of the static field. For example,

 private static int nextID; // all instances will agree on the next ID

Methods that are declared static do not have a calling object. Therefore, static methods perform a procedure without acting on an instance of their class. For example

public static boolean exceedsMaxTemperature(int temperature) {

 return temperature > MAX\_TEMPERATURE;

}

Topic 6a: Arrays

Because ArrayLists store collections of *objects*, it can be inconvenient to store *primitive* values (like 7 or -3.2 or true) in an ArrayList. Instead, it is better to use another Java data structure called an **array**. Arrays are also more efficient than ArrayLists for many operations. However, arrays can store only a fixed number of elements (whereas it is easy to increase or decrease the size of an ArrayList). To summarize:

* use arrays for *fixed size* collections
* advantages of arrays:
	+ can store either *primitive values* or object references
	+ accessing elements is more efficient than ArrayList
* disadvantage of arrays:
	+ cannot add or remove elements

To declare an array, use a pair of square brackets [] after the data type you would like to store in the array. For example,

 double[] prices;

declares a reference prices that will point to an array of double values. To actually create the array, use the new operator and specify the size of the array between the square brackets. For example, after the above declaration you could use

 prices = new double[3];

This creates an array that holds 3 elements of type double.

As with ArrayLists, the elements of an array are stored in a fixed order, and the position of an element in this order is called its index. The smallest index is 0, not 1.

To get the value of an array element, use the name of the array followed by square brackets with the desired index inside. For instance,

 double myPrice = prices[2];

stores the value of element 2 of the prices array in the local variable myPrice.

To set the value of an array element, use an ordinary assignment statement whose left-hand side is the name of the array followed by square brackets with the desired index inside. For instance,

 prices[1] = 78.33;

sets the value of element 1 of the prices array to 78.33.

You can combine the getting and setting of array elements in a single statement, such as

 prices[0] = 2\*prices[1] + prices[2];

The length or size of an array is obtained by using .length after the array name. For example,

 int len = prices.length;

Note that length is a field, not a method, which is why it is not correct to use parentheses – so prices.length() is wrong.

The elements of an array can be searched or altered in the same ways as for ArrayLists.

Topic 6b: nested loops

When a loop appears inside the body of another loop, the resulting code is called a **nested loop**. The two loops in a nested loop are called the **inner loop** and the **outer loop**. In a nested loop, all iterations of the inner loop body are executed for every single iteration of the outer loop. For example, here is a nested for loop:

 for (int i = 0; i < 3; i++) {

 for (int j = 0; j < 2 ; j++) {

 System.out.println(i + "," + j);

 }

 }

This produces the output:

0,0

0,1

1,0

1,1

2,0

2,1

The outer loop iterates over i, and the inner loop iterates over j. The inner loop executes 2 iterations every time the outer loop body is executed, and the outer loop executes 3 iterations, so the System.out.println statement is executed 2x3=6 times in total.

Topic 6c: Array Processing

For more complex array processing problems, it is important to take a systematic approach:

* Using English or pseudo-code, develop a sequence of steps that solve the problem.
* Try the steps by hand on a simple set of data.
* Translate those steps into Java.

Try to think of all special cases and test those by hand also. Important special cases can include:

* the array is empty
* the array contains only one element
* the first two values are identical
* the last two values are identical

Topic 6d: two-dimensional arrays

A two-dimensional array allows its elements to be accessed via two different indexes, usually thought of as rows and columns. Two-dimensional arrays are declared and initialized as in the following example:

String[][] labels = new String[5][3]; // five rows and three columns of String objects

Getting and setting the array elements works as you would expect:

 labels[1][2] = "apples"; // set element in row one, column two

 String fruit = labels[1][2]; // get element from row one, column two

Each row of a 2D array is in fact a one-dimensional array object. Therefore, it is possible to create **ragged** 2D arrays, whose rows are different lengths. For example, here is a ragged array with three rows of length two, four, and three respectively:

int[][] ragArray = new int[3][];

ragArray[0] = new int[2];

ragArray[1] = new int[4];

ragArray[2] = new int[3];

Topic 6e: the primitive types

Java’s eight primitive data types are

* **byte:** an integer value from -128 to +127
* **short:** an integer value in the range approximately ±32000.
* **int:** an integer value in the range approximately ±2 billion.
* **long:** an integer value in the range approximately .
* **float:** a floating-point (i.e. can be fractional) value with limited precision (i.e. only a few decimal places). Because the limited precision can easily produce numerical errors, float is only used when it is essential to save memory.
* **double:** a floating-point (i.e. can be fractional) value with twice as much precision as float. Numerical errors are still possible, but less likely to be significant.
* **boolean:** true **or** false**.**
* **char: used for storing a single ASCII or Unicode character, such as 'a', '?', or 'か'.**

Important warnings:

* All of the integer types can **overflow** without any exception being thrown. If overflow might occur, use java.math.BigInteger, which permits integers of arbitrary size.
* The two floating-point types accumulate inaccuracies due to rounding. For perfectly accurate calculations with decimal numbers, use java.math.BigDecimal.

Topic 7a: interfaces

In Java, an **interface** is used to specify a set of method signatures that other classes may choose to implement. In this way, an interface defines a kind of behavior that we can expect from any object that implements the interface. For example: the java.lang.Cloneable interface requires a clone() method that makes a new copy of the calling object; the java.lang.Readable interface requires a read() method that can read from a source of characters.

Programmers can define a new interface in a similar way to defining a new class. The interface source code is placed in a .java file whose name is the same as the interface, and the method signatures are listed as in the following example. Note that the methods do not have a body­—the methods are unimplemented.

public interface MakesSound {

 void makeSound();

 int howLoud();

}

Programmers can create a class that implements an interface by using the **implements** keyword, as in the following example:

public class Duck implements MakesSound {

 // fields, constructors, and methods for Duck omitted

 ...

 // implementation of the MakesSound interface

 public void makeSound() {

 System.out.println("Quack, Quack");

 }

 public int howLoud() {

 return 4;

 }

}

It is possible for a class to implement multiple interfaces, for example

public class Duck implements MakesSound, Swims { ...

Topic 7b: polymorphism for interfaces

In object-oriented programming languages, **polymorphism** is the idea that objects of different types can behave as if they have the same type in some circumstances. For example, if the Duck class and Sheep class both implement the MakesSound interface, then instances of both classes can be assigned to references of the type MakesSound:

MakesSound duck = new Duck();

MakesSound sheep = new Sheep();

Another way of looking at polymorphism is that a single object, such as duck in the above example, can simultaneously possess two or more different data types, switching between them as needed while the program runs:

duck.getSpecies() // behaving like a Duck

duck.makeSound() // behaving like MakesSound

A **polymorphic algorithm** is an algorithm that can process objects of different types. This can be achieved, for example, if the objects all implement the same interface.

Topic 7c: type casting and **instanceof**

In Java, a **type cast** can sometimes be used to convert an object from one type to another. A type cast is executed by placing the destination type in parentheses before the value to be converted, as in the following two examples:

Duck duck = new Duck();

MakesSound m = (MakesSound) duck;

double x = 5.8;

int y = (int) x; // y equals 5

In the first example, the type cast succeeds because of polymorphism: the duck object implements the MakesSound interface. The second example is rather different, because it involves a conversion of primitive types rather than object types. Conversions like this are built into Java and have certain pre-specified behavior. In this particular case, conversion from double to int truncates any fractional part, so the resulting value of y is 5.

Due to polymorphism, it is sometimes useful to test whether an object of one type is also an instance of—or implements—another type. In Java, this is achieved using the instanceof keyword, as in the following example:

 public static boolean isDuck(MakesSound m) {

 if (m instanceof Duck) {

 return true;

 } else {

 return false;

 }

 }

Some type casts are performed automatically by the Java compiler and need not be written out explicitly:

 Duck d = new Duck();

 MakesSound ms1 = (MakesSound) d; // unnecessary type cast

 MakesSound ms2 = d; // automatic type cast

 int x = 5;

 double y1 = (double) x; // unnecessary type cast

 double y2 = x; // automatic type cast

Some type casts are **illegal** and cause a **compiler error**:

 Sheep sheep = new Sheep();

 sheep = (Duck) d; // compiler error

The second line above results in the compiler error “Type mismatch: cannot convert from Duck to Sheep”.

Some type casts are not illegal yet lead to a **runtime error**:

 Duck d = new Duck();

 MakesSound ms = d;

 Sheep s = (Sheep) ms; // runtime error

This code compiles without error. But when it is executed, the last line throws a ClassCastException, with the associated error message “Duck cannot be cast to Sheep”.

Topic 7d: HashMap

Both in computer programs and in real life, you often need to use one piece of data to find out another piece of data. For example, when using a telephone book you start with one piece of data (a person's name, such as “Smith”) and use it to find out another piece of data (the person's telephone number, such as “727-347-2387”). The piece of data you start with is called the **key**, and the piece of data you end up with is called the **value**. Taken together, both pieces of data are called a **key-value pair**.

In Java, you can store a collection of key-value pairs using a HashMap. A HashMap constructor creates an empty collection of key-value pairs, and specifies the data types of the keys and the values using angled brackets, as follows:

 HashMap < keyType, valueType > ()

For example,

HashMap <String, Account> accountMap = new HashMap <String, Account> ();

declares and initializes a reference accountMap, which is a HashMap with keys of type String and values of type Account. The intention of this HashMap is that each key is the name of a person (e.g. “Smith”), and the corresponding value is a reference to that person's bank account.

Two important methods of the HashMap class are

* put(key, value)
	+ inserts a new key-value pair into the collection, with the given key and value
* get(key)
	+ returns a reference to the value associated with the given key

For example, the code

 Account account1 = new Account();

 accountMap.put("Smith", account1);

inserts a key-value pair into the HashMap, indicating that Smith’s bank account is account1.

The code

String name = "Smith";

Account theAccount = accountMap.get(name);

System.out.println("Balance for " + name + " is " +

theAccount.getBalance() );

first retrieves the account corresponding to “Smith” from the HashMap, and then prints the balance of this account.

The keys and values of a HashMap must have object data types—in other words, the keys and values may not have primitive data types such as int.

The HashMap class is not automatically available and must be imported using the statement

import java.util.HashMap;

Topic 7e: The Map interface

The Java HashMap class implements a mapping from keys to values using a data structure known as a **hash table**. We do not study the details of hash tables in this course. However, it is important to note that, in addition to hash tables, there are other ways of implementing a mapping from keys to values. For this reason, Java defines a Map<,> interface that can be implemented by other classes, including HashMap. If desired, you can declare your maps using this interface, as in the following example.

Map<String, Account> accountMap = new HashMap<String, Account> ();

Topic 7f: Set and HashSet

The most basic kind of collection in Java is a **Set**. When declaring a Set, use angled brackets to specify what type of objects the Set collection will contain. For example,

 Set<Account> accountSet;

declares a reference accountSet that will point to a set of Account objects.

The HashMap class has a method called keySet() that returns the set of all keys in the HashMap as a Set. For example, if a HashMap has been declared using the statement

HashMap <String, Account> accountMap = new HashMap<String, Account>();

then the statement

Set<String> nameSet = accountMap.keySet();

creates a Set storing all the keys in accountMap—in this case, the set of all names of people who have bank accounts in the HashMap. This is particularly useful for iterating over the keys of a Map.

In fact, Set is a Java interface that can be implemented in many ways. One concrete implementation is HashSet, which uses a hash table for storing the elements of the set. For example:

Set<String> nameSet = new HashSet<String>();

Topic 8a: inheritance

As we know, a **class** is a description of all possible objects with the same type of data and the same list of actions. A **subclass** of class is a subset of ’s objects, defined by the property that all objects in have some extra data and/or actions. If is a subclass of , we say is a **superclass** of .

Consider this concrete example. The software for a school district may include an Employee class with fields such as name, address, and salary. The Employee class could have a subclass called Teacher, with additional fields such as schoolName and currentCourses. The Teacher class is indeed a subset of the Employee class, because every teacher is an employee, but not every employee is a teacher. In this sense of being a subset, the Teacher class is *smaller* than the Employee class. On the other hand, every instance of the Teacher class is *larger* than a generic instance of the Employee class. This is because any Teacher object has extra data such as schoolName and currentCourses, in addition to the data such as name, address, and salary possessed by every Employee object. This distinction between a subclass being smaller and subclass *instances* being larger leads to apparently contradictory terminology: we say that a subclass is **derived from** its superclass, and we also say that a subclass **extends** its superclass. The word **inheritance** refers to the child-parent relationship between a subclass and its superclass, since a subclass inherits the fields and methods of the superclass.

In Java, we declare a subclass using the **extends** keyword, as in the following example.

public class Teacher extends Employee { ... }

Topic 8b: subclass constructors and super()

In the constructor for a subclass, we often wish to invoke the constructor of the superclass before performing other initialization specific to the subclass. This is achieved using the special built-in method super():

 public Teacher(String name, String address,

double salary, String schoolName) {

 super(name, address, salary);

 this.schoolName = schoolName;

 this.currentCourses = new ArrayList<String>();

 }

Topic 8c: overriding methods

A subclass can override methods that are provided by the superclass. To override a method, the subclass simply provides its own implementation of that method. When the method is invoked on an instance of the subclass, the subclass’s implementation will be executed. Example:

public class Employee {

 public void greet() { System.out.println("hi"); }

}

public class Teacher extends Employee {

 // override superclass method

 public void greet() { System.out.println("hello"); }

 public static void main(String[] args) {

 Teacher teacher = new Teacher();

 teacher.greet(); // prints "hello", not "hi"

}

}

Sometimes it is necessary to invoke a method from the superclass that has been overridden in the subclass. The superclass method can be invoked using the syntax super.method():

 // override superclass method

 public void greet() {

 super.greet();

 System.out.println("hello");

 }

It is important to understand the difference between overriding and overloading. When methods are *overridden*, they have the *same signatures* that occur in *different classes* (a subclass and superclass)—as in the above example. When methods are *overloaded*, they have *different signatures* but occur in the *same class*, as in the following example:

public int add(int a, int b) { return a + b;}

public int add(int a, int b, int c) { return a + b + c;}

Topic 8d: inheritance and polymorphism

Recall that polymorphism is the idea that objects of different types can behave as if they have the same type in some circumstances. Previously, we saw that interfaces give rise to polymorphism. Inheritance also leads to polymorphism, because instances of a subclass can also behave as instances of their superclass. Consider the continuation of the above Employee/Teacher example:

 Employee employee = new Teacher();

 employee.greet(); // prints "hello", not "hi"

The difference to the previous example is highlighted in bold. Even though the employee object is declared to be of data type Employee, it was constructed as an instance of the subclass Teacher. When the greet() method is invoked on the employee object, Java realizes that this object is really a Teacher, and the greet() method from the Teacher class is executed. The ability to execute a subclass method in this way is called **dynamic dispatch** or **dynamic method lookup**.

Topic 8e: inheritance and type casting

Just as with interfaces, the polymorphism that results from inheritance sometimes requires type casting, as in the following example:

public static void maybeGrade(Employee employee) {

 if(employee instanceof Teacher) {

 Teacher teacher = **(Teacher) employee**;

 teacher.doGrading(); // doGrading() is in Teacher but not Employee

 }

}

Topic 8f: inheritance from Object

Every class in Java automatically extends the built-in class Object. This means it is possible to write polymorphic algorithms that process objects of any type. It also means that all Java objects inherit Object methods such as toString(), equals(), and hashCode(). It is relatively common to override these methods.

We do not study equals() and hashCode() in detail, but it is easy to generate these methods automatically within Eclipse, and this is an essential technique for writing classes that can be used in HashMaps and HashSets.

The toString() method controls the output when an object is automatically converted to a string, such as in a call to System.out.println().

Topic 8g: protected

Recall that private and public are access modifiers, used to enforce information hiding. The keyword **protected** is another access modifier, enabling an intermediate level of information hiding. A field or method that is protected can be accessed by subclasses of the class in which it is declared, but not by any other classes. Experienced software developers sometimes disagree on the correct usage of protected, but there are some situations in which it is clearly advantageous.

Topic 8h: abstract classes and methods

In a Java class, it is possible to declare a method’s signature but omit the implementation. An unimplemented method is called an **abstract method**. A class with one or more abstract methods is called an **abstract class**. Abstract methods and classes must be declared using the abstract keyword:

public abstract class Bird {

 public abstract boolean canFly();

}

A class that is not abstract is called a **concrete class**. An abstract class can be extended to a concrete class by implementing all its abstract methods:

public class Penguin extends Bird {

 public boolean canFly() {

 return false;

 }

}

Abstract classes are similar to interfaces. The two important differences are: (i) an abstract class can have fields and method implementations; and (ii) a class can implement multiple interfaces, but it can extend only one superclass.

Topic 9a: Catching exceptions with try/catch

When Java code causes an exception to be generated, we say that it **throws** an exception. Consider the following example code.

public class Divide {

 private int numerator;

 public Divide(int numerator) {

 this.numerator = numerator;

 }

 public int divide(int denominator) {

 int result = numerator / denominator;

 return result;

 }

 public static void main(String[] args) {

 Divide d = new Divide(36);

 int x = d.divide(4); // x==9

 int y = d.divide(0); // throws java.lang.ArithmeticException

 }

}

The last line of code here causes an exception to be thrown because of division by zero. In fact, the exception is initially thrown in the first line of the divide() method. But an exception always propagates up the call stack, searching for a method which catches the exception. In Java, we can catch an exception using a try/catch block:

 // Exception example 1

 public static void main(String[] args) {

 try {

 Divide d = new Divide(36);

 int x = d.divide(4); // x==9

 int y = d.divide(0); // throws java.lang.ArithmeticException

 } catch (Exception e) {

 System.out.println("Sorry, an exception was generated: " + e.getMessage());

 }

 }

Exceptions can be caught anywhere in the call stack, but usually it is preferable to catch them as soon as possible:

 // Exception example 2

 public int divide(int denominator) {

 int result;

 try {

 result = numerator / denominator;

 } catch (Exception e) {

 System.out.println("Warning: exception was thrown");

 result = 0;

 }

 return result;

 }

Topic 9b: throwing exceptions and the throws clause

The programmer can cause an exception to be thrown by using the **throw** keyword followed by a newly constructed exception object:

 // Exception example 3

 public int divide(int denominator) throws Exception {

 if (denominator==0) {

 throw new Exception("Denominator cannot be zero.");

 }

For most types of exceptions, the Java compiler applies the *Catch or Specify Requirement*. This means the exception must either be caught via try/catch or specified via a throws clause, as in the above example.

Topic 9c: exception class hierarchy

The Java.lang.Exception class lies at the root of a class hierarchy, consisting of hundreds of subclasses which are built into the standard Java libraries. These include some commonly seen exception types such as NullPointerException and FileNotFoundException. It is good programming practice to throw and catch the most specific type of exceptions possible within this hierarchy. Thus, instead of the code labeled “exception example 1” above, the following would be preferable.

 // Exception example 4

 public static void main(String[] args) {

 try {

 Divide d = new Divide(36);

 int x = d.divide(4); // x==9

 int y = d.divide(0); // throws java.lang.ArithmeticException

 } catch (**ArithmeticException** e) {

 System.out.println(

"Sorry, an ArithmeticException was generated: "

+ e.getMessage());

 }

 }

Exception example 2 above should also catch an ArithmeticException, and Exception example 3 should throw an ArithmeticException.

It is also possible for the programmer to define new exception types by extending Exception or an appropriate subclass:

public class CoffeeException extends Exception { ...

A try/catch block can catch more than one exception type:

try {

 drink(description);

} catch (CoffeeException e) {

 System.out.println("CoffeeException: " + e.getMessage());

} catch (NullPointerException e) {

 System.out.println("NullPointerException: " + e.getMessage());

}

When there are multiple catch clauses, only the first clause that matches the thrown exception is executed. Here, “matches” means that the thrown exception is an instance of the type being caught.

9d: Checked vs Unchecked exceptions

There are the two main kinds of exceptions: checked exceptions, and unchecked exceptions. Most exceptions are **checked** exceptions; the exception type RuntimeException and its subclasses are **unchecked**. The *Catch or Specify Requirement* described above applies to all checked exceptions. In contrast, the requirement is not applied to unchecked exceptions. The philosophy behind this is that the programmer should handle checked exceptions in a thoughtful way, because checked exceptions arise from conditions that the programmer can anticipate. For example, the NoSuchFileException exception is checked because the programmer should always anticipate that a given file name may not correspond to a file that exists. In contrast, exceptions that are subclasses of RuntimeException are not required to be handled because they should not arise in the first place. For example, NullPointerException is a subclass of RuntimeException, and is therefore unchecked. This is because in a well-written program, a NullPointerException never occurs. If a NullPointerException is thrown, this is a bug that the programmer must fix.

9e: Testing for Exceptions

When testing code that can throw a checked exception, the test methods must achieve code coverage as usual. In particular, this means that one of the tests should trigger the exception, catch it, and assert that the expected type of exception was thrown:

 public void testDrink() {

 Coffee coffee = new Coffee();

 try {

 coffee.drink("nice and strong");

 } catch (CoffeeException e) {

 fail("Strong coffee threw an exception");

 }

 boolean caughtException = false;

 try {

 coffee.drink("subtle");

 } catch (CoffeeException e) {

 caughtException = true;

 }

 assertTrue("Failed to catch CoffeeException", caughtException);

 }

Topic 9f: finally

Sometimes, objects consume external resources that need to be freed up before they go out of scope. One common example is that an object may open a file or stream which should be closed before the object’s lifetime comes to an end. Another example would be an object that creates a temporary file which should be deleted before the object’s lifetime comes to an end. In such cases, the programmer can add a finally clause to a try/catch statement. The finally clause is guaranteed to be executed, whether or not an exception was caught:

 Coffee coffee = new Coffee();

 coffee.createTemporaryFile();

 try {

 // Throws a NullPointerException, which won't be caught

 coffee.drink(null);

 } catch (CoffeeException e) {

 System.out.println("Exception: " + e.getMessage());

 } finally {

 coffee.deleteTemporaryFile();

 }

Topic 10a: recursive mathematical functions

In mathematics, a function is **recursive** if the value for a given input is defined in terms of the values for smaller inputs, such as or . For example, if represents the sum of the first even positive integers, then can be defined as follows:

The first line above is called the **base case**, and the second line is called the **recursive case**. It is possible to have multiple base cases and multiple recursive cases.

Topic 10b: Tracing recursive executions

For small values of the input, we can manually **trace** the evaluation of a recursive function, as in this example which uses the above definition of :

10c: recursive Java methods

To compute a recursive function in Java, use if/else to separate the base case(s) and recursive case(s); and inside each recursive case, invoke the recursive method where appropriate:

 // Compute the sum of the first n even integers

 public int sumOfEvenIntegers(int n) {

 if (n == 1) {

 return 2;

 } else {

 return 2 \* n + sumOfEvenIntegers(n - 1);

 }

 }

10d: Thinking recursively; recursive problem transformation

The key idea behind **thinking recursively** is to solve a problem using a simpler version of the same problem. When seeking a recursive solution, follow these steps:

1. Assume you can solve a simpler version of the problem.
	* Don’t worry about how to solve the simpler version; just assume you already have the answer to it.
2. Solve the original problem by using the answer to the simpler version.

Often, the original form of a problem is not perfectly suited to a recursive solution. In such cases, we apply **recursive problem transformation**. There is no fixed method for doing this, but it often involves adding extra parameters to the recursive case so that the recursive function has more information to use for solving the problem.

Mini topic: refactoring

The Refactor menu in Eclipse provides many useful shortcuts for refactoring code.

Mini topic: command line arguments in main()

Recall that the main method has the following signature:

public static void main(String[] args)

Here, the parameter args is an array of strings known as **command line arguments**. These can be used to provide input to a Java program. In Eclipse, set command line arguments by selecting Run | Run Configurations… | Arguments | Program arguments.

Each argument is separated from the previous argument by one or more spaces. The arguments are passed to the main method as Strings in the array args, with args[0] being the first command line argument, args[1] being the second command line argument, and so on. A useful method when processing command line arguments is Integer.parseInt() which converts a String into an int. For example, the following main method for Account uses the first 3 command line arguments as the initial balance of a bank account, the initial interest rate, and the amount to withdraw from the bank account:

 public static void main(String[] args) {

 int initBalance = Integer.parseInt(args[0]);

 int initInterest = Integer.parseInt(args[1]);

 int withdrawAmount = Integer.parseInt(args[2]);

 Account account = new Account(initBalance, initInterest);

 account.withdraw(withdrawAmount);

 int newBalance = account.getBalance();

 System.out.println("New balance is " + newBalance);

 }

When given the command line arguments “3000 12 250”, this program creates a new bank account with balance 3000 and interest rate 12, withdraws 250 from the bank account then prints the string “New balance is 2750”.

Mini topic: compiling and running from the command line

It is possible to compile and run Java programs within a terminal window. Here we describe only the simplest possibility, assuming we have a single .java file, Account.java. We assume Account.java is in the default Java package (i.e. it has no package keyword).

To **compile a class** from the command line, the following steps are required:

* open a **terminal window** (Mac) or a **command prompt window** (Windows)
* change into the directory containing the Java program. Useful commands for doing this include:
	+ **pwd** (Mac) or **cd** (Windows) — print **current directory**, also called the **working directory**
	+ **ls** (Mac) or **dir** (Windows) — list current directory
	+ **cd** newDir — change directory into the newDir folder
	+ **cd** .. — change directory into parent folder
* enter the command javac ClassName.java, where ClassName is the name of the class you want to compile. For example, the command java Account.java would compile the Account class.

Compiling a .java file produces a corresponding .class file (e.g. Account.class), containing **Java byte code**. This byte code can be executed by the **Java virtual machine** (JVM).

To **run a program** with a main method from the command line, the following steps are required:

* open a terminal window and change into the program's folder as above
* enter the command java ClassName, where ClassName is the name of the class whose main method you want to run. For example, the command java Account would execute the above main method of the Account class.

The interface provided by a terminal window, where you type a single line of commands at a time, is often called a **command line**. When you run a program from the command line, you can specify command line arguments after the name of the class. Each argument is separated from the previous argument by one or more spaces. Continuing the above example of Account.java, consider the terminal command “java Account 3000 12 250”. This command creates a new bank account with balance 3000 and interest rate 12, withdraws 250 from the bank account then prints the string “New balance is 2750”.

Mini topic: design patterns

In software engineering, a **design pattern** is a way of structuring source code to suit a commonly occurring scenario.

The **singleton** design pattern is used to ensure that only one instance of a given class is ever created. This could be useful, for example, in a Log class that logs messages from every other class in an application. There are many ways of implementing the singleton design pattern. One approach employs the following two techniques:

1. The singleton instance is defined as a static field which can be accessed via a public getter.
2. The constructor is declared private to prevent external code creating new instances.

Here is an example of the approach:

public class Log {

 // singleton instance

 private static Log THE\_LOG = new Log();

 private ArrayList<String> messages;

 // constructor is private so external code cannot create new Log instances

 private Log() {

 messages = new ArrayList<String>();

 }

 public static Log getLog() {

 return THE\_LOG;

 }

 …

The **adapter** design pattern is used when we have a class that must be adapted for use in a situation that was not originally envisaged when the class was written. Specifically, suppose we have a class , which is part of an external library and whose source code cannot be edited. We are writing a program that processes objects of various classes, say , which all implement some standard interface or provide some standard public methods, say . We would like to be capable of also processing instances of , but does not implement the methods . This is where we use an adapter: we create a **wrapper** class which has a field of type . We then implement the methods in , making appropriate calls to .

In the following example, Sheep is the external library class ; AdaptedSheep is the wrapper class ; getVolume() is the desired method .

public class Sheep implements MakesSound {

 ...

 // We need a getVolume() method,

// but we only have a howLoud() method

 public int howLoud() {

 return 4;

 }

 ...

}

public class AdaptedSheep {

 private Sheep sheep;

 public AdaptedSheep(Sheep sheep) {

 this.sheep = sheep;

 }

 // Adapt howLoud() to getVolume()

 public int getVolume() {

 return sheep.howLoud();

 }

...

}

Topic 11a: selection sort

The selection sort algorithm sorts an array of values as follows:

1. Find the smallest value in locations , and swap it into location 0.
2. Find the smallest value in locations , and swap it into location 1.
3. Find the smallest value in locations , and swap it into location 2.
4. …

This can be implemented in Java with two helper methods: (i) a method that finds the smallest value in a given set of locations, and (ii) a method that swaps two locations.

Topic 11b: Asymptotic analysis and big O notation

When analyzing the efficiency of algorithms, the *absolute* amount of time required to run an algorithm is often less relevant than the *shape* of the curve produced when we make a graph of the running time as a function of the input length. This type of analysis is called **asymptotic analysis**: we focus on the *dominant* shape of the *worst-case* running time curve for *large* values of the input length, ignoring *constant factors*. The mathematical notation for asymptotic analysis is known as **big O notation**. Common examples of big O notation are given in the following table, where denotes the length of the input.

|  |  |  |  |
| --- | --- | --- | --- |
| **big O notation** | **Running time is…** | **Running time is proportional to at most…** | **Practical effect** |
|  | **constant** | a constant | When increases, running time does not change |
|  | **logarithmic** |  | When doubles, a fixed amount is added to the running time. |
|  | **linear** |  | When doubles, the running time doubles. |
|  | **log-linear** |  | When doubles, the running time doubles and a fixed amount is added. |
|  | **quadratic** |  | When doubles, the running time increases by a factor of 4. |
|  | **exponential** |  | When increases by a fixed amount, the running time doubles. |

When expressed in big O notation, the running time of an algorithm is called its **asymptotic running time**. The asymptotic running times in the table above are listed in order from best to worst. The green running times are considered acceptable; the orange (quadratic) is poor but sometimes acceptable; the red (exponential) is almost always unacceptable except for very small inputs.

When converting from an absolute running time to big O notation, discard all terms except the largest one (according to the above list), and also discard any constant multiplicative factors. For example, a running time of becomes ; a running time of becomes .

The asymptotic running time of an algorithm is also known as its **computational complexity**, or just its **complexity**.

Topic 11c: Asymptotic analysis of selection sort

When computing asymptotic running time, we need to agree on which **elementary operations** will count towards the running time.

* For sorting algorithms, we usually count the number of **comparisons**: that is, the number of times any element of the array to be sorted was compared against some other element.
* For the searching algorithms considered later, we usually count the number of **array accesses**: that is, the number of times any element in the array to be searched was either read or written.

The precise choice of which elementary operations to count does not usually affect the asymptotic running time. This is because the asymptotic running time is concerned with the overall shape of the running time curve, ignoring the multiplicative scale and other insignificant factors.

From the description of selection sort given above, we can see that there are approximately steps that involve “find the smallest value”. Each of these steps requires at most comparisons to find the smallest value. Thus, we have at most comparisons. Hence this algorithm is in .

Topic 11d: Merge sort

To understand merge sort, we first consider an essential subroutine of the algorithm, which merges two sorted lists into a single sorted list. This subroutine, which we will call *merge-sorted-lists*, is easy to visualize if we imagine two piles of cards. Each card has a single number printed on it, and each pile of cards is sorted into increasing numerical order. To produce a single sorted pile of cards, we choose the smaller of the two top cards and put it in a new pile. We repeat this process until all the cards are in the new pile. If the total number of cards is , this merge-sorted-lists subroutine runs in time .

Now we can state the full merge sort algorithm. We begin with a pile of (unsorted) cards. We sort this pile using a recursive approach with the following base case and recursive case:

* Base case: If the pile has only one card in it, it is sorted, and we are done.
* Recursive case: The pile has at least two cards, so split it into two piles that each contain half of the cards. (If there is an odd number of cards, split the pile as equally as possible; the algorithm still works.) Then:
	+ Apply the merge sort algorithm separately to each of the two new piles. (This is the recursive application of merge sort.)
	+ Run the merge-sorted-lists subroutine to produce a single sorted pile from the two sorted piles.

In this course, it is not essential to understand fully the asymptotic analysis of merge sort. It is essential to memorize the result: merge sort is an algorithm—that is, log-linear. For completeness, we now briefly investigate why merge sort is log-linear. The number of times that the piles are split into smaller piles is . We can think of these splits occurring in *levels*, where each level has piles half as big as the previous level. For each of these levels, the merge-sorted-lists subroutine is run separately on several pairs of piles, but in every case the total number of cards in the piles at that level is As noted above, merge-sorted-lists costs , and this cost is incurred at each of the levels. Thus, the total cost is , which simplifies to

There are several other well-known, efficient sorting algorithms, including *quick sort* and *heap sort*. Their computational complexity is similar to merge sort.

Topic 11e: linear search

Although it was not emphasized earlier, the definition of big-O notation includes the words “worst case.” Informally, this translates as “at most” or “at worst” when interpreting asymptotic running times. For example, if an algorithm is in , then its running time is guaranteed to be *at most* proportional to . This type of analysis is called **worst case** analysis, because we state an upper bound for the worst possible running time that could occur.

Suppose we are given a list , containing items. For simplicity, assume the items are strings, although this discussion would apply to any type of item. We are given a string , and our task is to search for an occurrence of in . If an occurrence of is found, the index of in is returned (e.g. 0 if it is the first item, if it is the last item). If no occurrence of is found, the special value *None* is returned.

We can solve this problem using the following algorithm, which is called **linear search**.

* Examine the first item. If it equals , return 0.
* Examine the second item. If it equals , return 1.
* …
* Examine the last item. If it equals , return .
* Return *None*.

To calculate the asymptotic running time of linear search, first recall that we will count array accesses as our elementary operations. In the worst case here, the algorithm accesses each element of the array exactly once, resulting in array accesses. Hence, the complexity of linear search is .

Topic 11f: binary search

Consider the same search problem as earlier: we are given a list , containing strings or other items that can be compared. We are given a string and our task is to return the index of an occurrence of in , or the special value *None* if is not present. However, now we assume the list is *sorted* in increasing order (e.g. dictionary order for strings or numerical order for numbers). When the list is sorted, we can use the following algorithm, known as **binary search**, to efficiently narrow in on the location of .

* The algorithm maintains a lower bound (for Bottom) and an upper bound (for Top) on the index of in . Initially, and . The algorithm ensures that if is present at all, it is always at a location between and . The exact property maintained is that .
* Repeat the following steps until :
	+ Compute the midpoint between and using integer arithmetic. Call this midpoint . So , rounded down if necessary.
	+ Examine the item at index in . Call this item .
	+ If , assign the new value .
	+ If , assign the new value .
* At this point, the numbers all have the same value, which we will now call . If the element at location equals , we return . Otherwise, we return *None*.

To calculate the computational complexity of binary search, observe that the repeated “narrowing-in” loop dominates all other operations. The running time will be proportional to the number of times that this loop is executed. In other words, we need to know how many times we can halve the distance between and until they have the same value. This is easier to analyze in reverse: how many times would we need to double the distance between and until they are separated by their initial difference, ? To put it even more simply, how many elements are in the sequence ? (Ignore the fact that may not be a power of 2. With asymptotic analysis, we can ignore the fact that the number of elements in our sequence may be off by one.) By the definition of the log function, the number of elements in the sequence is (where we again accept a slight approximation if is not a power of 2). Hence, the computational complexity of binary search is .

Note that binary search is far more efficient than linear search: versus . However, binary research requires that the list is first sorted. If the list is not already sorted, it would require to sort it via merge sort or another sorting algorithm.

Note that binary search can be implemented in an elegant fashion using recursion, as in the provided file Search.java.

Topic 11g: Java library searching and sorting algorithms

The java.util.Collections package contains several polymorphic algorithms for searching and sorting collections. These include binarySearch(List<T> list, T key) and sort(List<T> list).

Note that List<T> is an interface, implemented by ArrayList<T>.

Topic 11h: comparable interface

Any Java class that implements the Comparable interface possesses a **natural ordering**. This means that, given any two instances of the class, we can compare the two instances and obtain exactly one of three results: either is greater than , or is less than , or is equal to . This comparison is implemented by the **compareTo()**method of the Comparable interface. By definition, A.compareTo(B)returns an integer that is

* positive, if is greater than
* negative, if is less than
* zero, if is equal to

It is often easiest to implement compareTo()by invoking it on one or more fields of the current class. For example, the following code sorts instances of the SuperHero class by their name field.

public class SuperHero implements Comparable<SuperHero> {

 ...

public int compareTo(SuperHero other) {

 return name.compareTo(other.name);

}

}

The sort(List<T> list) method in java.util.Collections sorts the given list according to the natural ordering:

 ArrayList<SuperHero> heroes = ...

 Collections.sort(heroes);

Topic 11i: comparator interface

To sort a collection in an order that is not the natural ordering, the Comparator interface can be used. An implementation of the Comparator interface must implement the compare() method as in the following example:

public class HeightComparator implements Comparator<SuperHero> {

 public int compare(SuperHero arg0, SuperHero arg1) {

 return Double.compare(arg0.getHeight(), arg1.getHeight());

 }

}

The return value of compare() is defined in a similar fashion to the compareTo()method above: negative if arg0 is less than arg1, zero if they are equal, and positive otherwise. It is usually easiest to implement compare() by invoking it on one or more fields of the current class.

To sort a collection using compare(), we pass an instance of the relevant comparator as a second parameter in java.util.Collections.sort():

Collections.sort(heroes, new HeightComparator());

Topic 12a: the List abstract data type

An **abstract data type** (ADT) is a mathematical description of some information that can be stored and some operations that can be performed on that information. An ADT does not specify how the operations should be performed or how the data should be stored. It specifies the external behavior of the operations, but not the implementation. Thus, an ADT is similar to a Java interface or abstract class, except that an ADT is a purely mathematical concept that does not depend on any one programming language.

The **List ADT** describes an ordered collection of items. There are six operations defined for the List ADT:

* get the size of the list
* add an element to the end of the list
* get the th element of the list
* set the th element of the list
* insert an element before the th element
* remove the th element of the list

An ADT can be implemented by a **concrete data type**. A concrete data type does specify how the data should be stored, and how the operations should be performed. For example, the Java ArrayList is a concrete data type that implements the list ADT. However, there are other—very different—ways of implementing the list ADT. Later, we study the *linked list* as one example of this.

In Java, the List ADT is provided by the java.util.List interface.

Topic 12b: array-based implementation of the List ADT

An array-based list implements the List ADT by storing the items in a consecutive sequence of memory locations. We describe the implementation in a way that is mostly independent of which programming language is used, but we occasionally employee Java-specific details for concreteness.

The number of locations (or **slots**) available for storing items is called the **capacity** of the array. Usually, the initial capacity is a small constant such as 10. Items are always stored consecutively in the array starting from slot 0. The list operations are performed as follows.

* Get size: The current size is stored in a variable and can be returned whenever needed. Note that the size is not the same as the capacity. The size is the number of elements currently stored, which is always less than or equal to the capacity.
* Add to end: If the size is less than the capacity, store the element to be added in the next available slot. If the size is already equal to the capacity, create a new array with double the capacity and copy all existing elements into this array before adding the new element. In all cases, update the size variable.
* Get th element: Return the element in slot number .
* Set th element: Overwrite the element in slot number .
* Insert before th element: Starting with the last element and working backwards to the th element, move each element up one slot. This leaves a gap at location , whose value can be set to the new element. Note: If the size is already equal to the capacity, first create a new array with double the capacity and copy all existing elements into this array before adding the new element. This is the same subroutine as in the *add* operation.
* Remove th element: Starting with the th element and working forwards to the last element, move each element down one slot.

In Java, the array-based implementation of List is provided by java.util.ArrayList.

Topic 12c: linked list implementation of the List ADT

In a linked list, the elements of the list are stored in arbitrary memory locations, and **pointers** are used to determine the location of each element from the previous element. (In Java, the concept of *pointer* is implemented as an object reference. As we know, an object reference is really the address of an object in the computer’s memory.)

Each element is wrapped in a small data structure called a **node**. Each node contains its corresponding *element*, and a *next* pointer. The *next* pointer is a reference to the node containing the next element in the list. If there is no next element, the value of the *next* pointer is null. The **head** of the list is a special pointer that is stored as a variable and always points to the first node. The **tail** of the list is a special pointer that is stored as a variable and always points to the last node.

The list operations are performed as follows. These descriptions are deliberately informal and the special cases for dealing with changes to the head and tail nodes are not always addressed in full detail.

* Get size: The current size is stored in a variable and can be returned whenever needed.
* Add to end: Create a new node containing the element to be added and a null *next* pointer. Update the *next* pointer of the tailnode so that it points to . Update tail so that it points to . Update the size variable.
* Get th element: Start at the head node. Follow this node’s next pointer. Repeat this times in total. Return the element in the node that is reached.
* Set th element: Find the th node exactly as in the “get th element” operation. Then overwrite the element in the node that is reached.
* Insert before th element: Create a new node containing the element to be added. Find node in a similar fashion to the “get th element” operation. Let us denote this located node by . Set ’s *next* pointer to be the same as ’s *next* pointer. Update the *next* pointer of so that it points to . Update the size variable. If , update the head pointer so that it points to .
* Remove th element:
	+ If , find node and node in a similar fashion to the “get th element” operation. Let us denote the located node by and node by . Set ’s *next* pointer to be the same as ’s *next* pointer.
	+ If , update head so that it has the same value as the current head’s *next* pointer.
	+ Update the size variable.

In Java, the linked list implementation of List is provided by java.util.LinkedList.

Some of the operations in a linked list can be made more efficient by storing links to the previous node as well as the next node. This kind of data structure is known as a **doubly linked list**. The java.util.LinkedList is in fact doubly linked.

Topic 12d: Comparing ArrayList and linked list

The computational complexity of array-based and linked list operations are as follows. The number of items in the list is denoted by .

|  |  |  |
| --- | --- | --- |
| operation | array-based list | linked list |
| Get size |  |  |
| Add to end |  but on average |  |
| Get th element |  |  |
| Set th element |  |  |
| Insert before th element |  |  |
| Remove th element |  |  |
| Insert or remove first element |  |  |
| Traverse using get() for  |  |  |

Topic 12e: iterators

Recall that the foreach loop provides an elegant and compact way of iterating over collections. We would like to be able to use foreach loops on classes that we write ourselves. This can be achieved using the Iterator<> and Iterable<> interfaces. For example, suppose we would like to use foreach loops for iterating over the friends stored in the following Friends class:

public class Friends implements Iterable<String> {

 private ArrayList<String> friendList;

 public String getFriend(int i) {

 return friendList.get(i);

 }

…

}

Note that this class implements the Iterable<String> interface. The type String is employed here because when we iterate over a Friends instance, the element produced at each iteration will be a String. The first step to achieve our goal is to create a new class implementing the Iterator<> interface:

class FriendsIterator implements Iterator<String> {

 private ArrayList<String> theFriendList;

 private int index = 0;

 public FriendsIterator(ArrayList<String> theFriendList) {

 this.theFriendList = theFriendList;

 }

 @Override

 public boolean hasNext() {

 return index < theFriendList.size();

 }

 @Override

 public String next() {

 String nextFriend = theFriendList.get(index);

 index++;

 return nextFriend;

 }

}

Notice how the FriendsIterator class implements the hasNext() and next() methods. There is also a field for remembering the current iteration.

The second and final step to achieve our goal is to go back to the Friends class and implement the iterator() method of the Iterable<> interface:

public class Friends implements Iterable<String> {

…

 @Override

 public Iterator<String> iterator() {

 return new FriendsIterator(this.friendList);

 }

…

}

It is now possible to iterate over the Friends class using a foreach loop as desired:

 Friends friends = new Friends();

 …

 for (String friend : friends) {

 System.out.println(friend);

 }

This may seem like a rather small gain for such a large amount of work. But it is in fact surprisingly useful. In addition, there are other important uses of iterators that lie beyond the scope of this course.

The accompanying file FriendsNested.java has another interesting feature. Notice that the FriendsIterator class is not stored in a separate FriendsIterator.java file. Instead, the code for the FriendsIterator class appears *inside* the Friends class. This is called a **nested class**. When a small class  is used only by one other class , it is often a good idea to nest class  inside class . This usually results in code that is more readable and maintainable.