Chapter 1 – Classes & Objects

Contents

[1 Introduction 3](#_Toc175920914)

[2 Class Modelling 4](#_Toc175920915)

[3 Instance Variables & Methods 5](#_Toc175920916)

[4 More on Classes 7](#_Toc175920917)

[4.1 Exercises 7](#_Toc175920918)

[5 Introduction to Constructors 8](#_Toc175920919)

[5.1 Exercises 8](#_Toc175920920)

[6 Instance Variables & Getters 9](#_Toc175920921)

[6.1 *private* Instance Variables 9](#_Toc175920922)

[6.2 Getters 10](#_Toc175920923)

[6.3 An Array Instance Variable & Indexed Getter 12](#_Toc175920924)

[7 Setters 13](#_Toc175920925)

[7.1 Indexed Setter 14](#_Toc175920926)

[7.2 Exercises 15](#_Toc175920927)

[8 The *this* Reference 16](#_Toc175920928)

[8.1 Exercises 16](#_Toc175920929)

[9 The *toString* Method 16](#_Toc175920930)

[9.1 Exercises 17](#_Toc175920931)

[10 Testing a Class 18](#_Toc175920932)

[10.1 Exercises 20](#_Toc175920933)

[11 Class & Object Diagrams 22](#_Toc175920934)

[12 Best Practices – Naming Conventions 23](#_Toc175920935)

[12.1 Exercises 24](#_Toc175920936)

[13 Best Practices – Writing Classes 24](#_Toc175920937)

[14 The String.format Method 25](#_Toc175920938)

[14.1 Exercises 27](#_Toc175920939)

[15 Helper Methods 28](#_Toc175920940)

[16 Accessing Objects via Reference Variables 29](#_Toc175920941)

[17 The null Keyword 31](#_Toc175920942)

[18 Using *this* With Multiple Constructors 32](#_Toc175920943)

[18.1 Exercises 33](#_Toc175920944)

[19 Passing Objects to Methods 33](#_Toc175920945)

[19.1 Exercises 35](#_Toc175920946)

[20 Arrays of Objects 36](#_Toc175920947)

[20.1 Exercises 38](#_Toc175920948)

[21 Immutability 39](#_Toc175920949)

[22 Data Encapsulation 41](#_Toc175920950)

[23 Packages & Accessibility (Visibility) of Classes & Members 42](#_Toc175920951)

[24 Static Variables & Methods 44](#_Toc175920952)

[24.1 Static Variables 44](#_Toc175920953)

[24.2 Constants – *final* 45](#_Toc175920954)

[24.3 Class Constants – *static final* 46](#_Toc175920955)

[24.4 Static Methods 46](#_Toc175920956)

[24.5 References 46](#_Toc175920957)

[25 Chapter Summary 47](#_Toc175920958)

[Appendix 1 String & Characters 49](#_Toc175920959)

[Appendix 1.1 Characters 49](#_Toc175920960)

[Appendix 1.2 Strings 49](#_Toc175920961)

[Appendix 1.3 String Comparison Methods 51](#_Toc175920962)

[Appendix 1.4 Substring Methods 53](#_Toc175920963)

[Appendix 1.5 Conversion of other Types to String 53](#_Toc175920964)

[Appendix 1.6 String Location Methods 53](#_Toc175920965)

[Appendix 2 The *BoxUtilities* Class 55](#_Toc175920966)

[Appendix 3 The Enhanced For Loop (aka The for-each Loop) 57](#_Toc175920967)

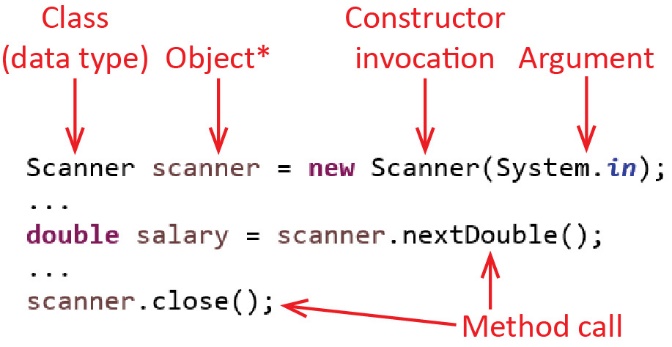
[Appendix 4 State Information for a Network Printer 59](#_Toc175920968)

To make this document easier to read, it is recommended that you turn off spell checking and grammar checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”

# Introduction

In CS 1301 you learned the basics of *programming*: variables, data types, selection, loops, methods, arrays, *etc*. to solve small problems. In addition, you also learned how to use a few classes from the Java API[[1]](#footnote-1) to help you solve problems: *Math, String,* and *Scanner*. For example, you used the *Scanner* class to read data from the console:



Review:

* The *scanner* object is created from the *Scanner* class’s *constructor*.
* The *scanner* object is used to provide services by having methods called on it.
* \*The *scanner* variable is not actually an object. It is actually a variable that holds a reference (pointer) to a *Scanner object* that exists elsewhere in memory. We will discuss this important distinction later.

To solve larger problems, a popular approach is to use *object-oriented programming (OOP* or just *OO)*. OOP means, in addition to using the basics of programming, and classes from the Java API, we also write our own, custom classes and use them to help solve problems.

Most of the first half of this course is course is concerned with writing and testing classes. Later in the course, when we cover graphical user interfaces (Gui), we will use these classes to solve problems.

# Class Modelling

OO problem solving is a little different than the approach you used in CS 1301. The first thing we do is *class modelling* which involves figuring out what classes we need to write. We do this by identifying the natural *objects* (entities, “things”) that are present in the *problem* *domain.* Objects are usually nouns. A few examples are shown below.

|  |  |
| --- | --- |
| **Problem Domain** | **Possible Objects** |
| Basketball Simulation | Player, Referee, Ball, Court, ScoreBoard, *etc* |
| Amazon | Product, Shopping Cart, Order, Account, Customer, *etc* |
| Banking System | Account, Customer, Employee, Bank, *etc* |
| University Registration System | Course, Class, Student, Professor, GradeBook, *etc* |

Next, to write an OO program, we need a way to represent these natural objects as entities in our program. We do this by defining a *class* for each type of object. A class is a formal Java construct. For example, the code below defines an *Account* class. It is not very useful yet, but it is a valid class.

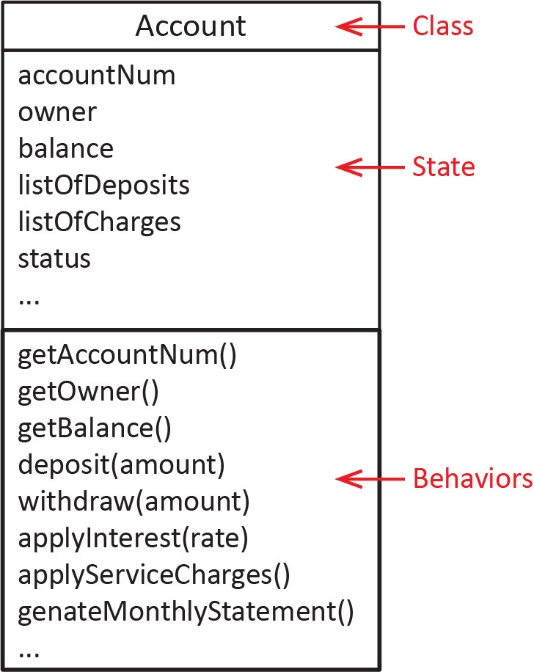
**public** **class** Account {

}

Each class is typically stored in its own file. The name of the file must be the same as the class with a *.java* file extension. For example, the *Account* class would be stored in a file named, *Account.java*.

So, what exactly is a class? A class is a way to represent:

* The *state* (the current conditions, data) that an object has
* The *behavior* (the things it can do) that an object has

Note that elements of the state are usually nouns, and behaviors are usually verb phrases. For example, consider an *Account* class that is used to model a bank account. Some elements of the state and behaviors might be:

**State** – Account number, Account owners name, current balance, list of deposits, list of charges (ATM, debit transactions, checks, *etc.*), status (good standing or overdrawn), *etc*.

**Behaviors** – Deposit funds, withdraw funds, apply interest, apply service charges, generate monthly statement, *etc*.

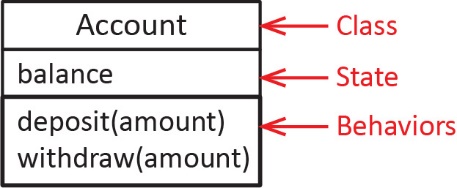
As shown in the figure on the right, we usually document class modelling with a *class diagram*. An important feature of a class is that it *encapsulates* the state and behaviors into a single unit, the *Account* class.

# Instance Variables & Methods

The code for this class is shown below (see *example\_account/Account1.java* in code download):

Consider this simplified description of an *Account* class:

Write an (checking) *Account* class that keeps track of the balance of the account. It provides a way to deposit an amount of money. If the amount is positive, then the balance is increased by this amount; otherwise, no change is made to the balance. Similarly, the class provides a way to withdraw an amount of money, which, if positive, decreases the balance by this amount.

Here, the state is simply the *balance* of the account, and two behaviors are: *deposit* and *withdraw* as shown in the class diagram on the right.

Next, we need to consider how we write code for state and behaviors?

* *State* (also called *properties* or *attributes*) is implemented with *instance variables* (also called *data fields* or *fields*)and are similar to the variables you have used in programs previously.
* *Behavior* (also called *services, responsibilities*) is implemented with *methods*.

**public** **class** Account {

**public** **double** balance;

**public** **void** deposit(**double** amount) {

**if**(amount>0) {

balance += amount;

}

}

**public** **void** withdraw(**double** amount) {

**if**(amount>0) {

balance -= amount;

}

}

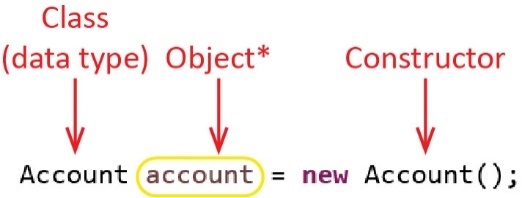
}

Note:

* The *balance* instance variable is the “memory” of the class, it remembers the value of the balance at all times.
* An instance variable is available anywhere in the class. For example, in the *deposit* method, the *balance* instance variable is updated by the parameter, *amount*.
* Everything is declared with *public visibility*. All Java entities (classes, methods, instance variables) can have a *visibility modifier* prefacing the declaration. *public* visibility means that the entity can be seen by other classes (as well as inside the class itself). We will address visibility several more times in this chapter.

A class can have data fields that are primitive data type or reference data types (*String* is the only one we know now). Primitive data types take on the default values of 0 for numeric types and *false* for boolean. Reference data types take on, by default, the special literal value, *null*, which we discuss in another [section](#_The_null_Keyword).

A class is a blueprint (or template) for creating objects and we use objects to write programs. For example:



Note:

* To create an object, we use a *constructor* preceded by the *new* keyword. Thus, a constructor’s job is to create an *object* (also called an *instance*).
* In the *Account* class above, we did not define (write) a constructor; however, Java provides one implicitly. We learn to write our own constructors shortly. The constructor above is called a *no-arg constructor* because it does not accept any arguments.
* We use the class (*Account*) as the datatype for the *account object.* The *account object* is really just a variable and so it must have a data type specified. We use the class of the object as the data type.

We can use the *account* object above to access the instance variables or methods by invoking the *dot operator*. The general syntax is:

* object.instanceVariable
* object.method(arguments)

For example, we could write a main with code like this:

**public** **static** **void** main(String[] args) {

Account account = **new** Account();

account.deposit(1000.0);

account.withdraw(400.0);

System.***out***.println("bal=" + account.balance); // 600.0

}

Some observations:

* A class is a way to group together related data and methods. This is called *encapsulation* meaning that we have one “unit”, the *Account* class that holds the data and methods.
* The methods do something with or to the data. For example, the *deposit* method changes the *balance* by adding *amount* (1000 in the example above)*.*
* The data is *inside* the object. For example, when we want the current balance, we access it by: account.balance.

# More on Classes

The three components of a class are:

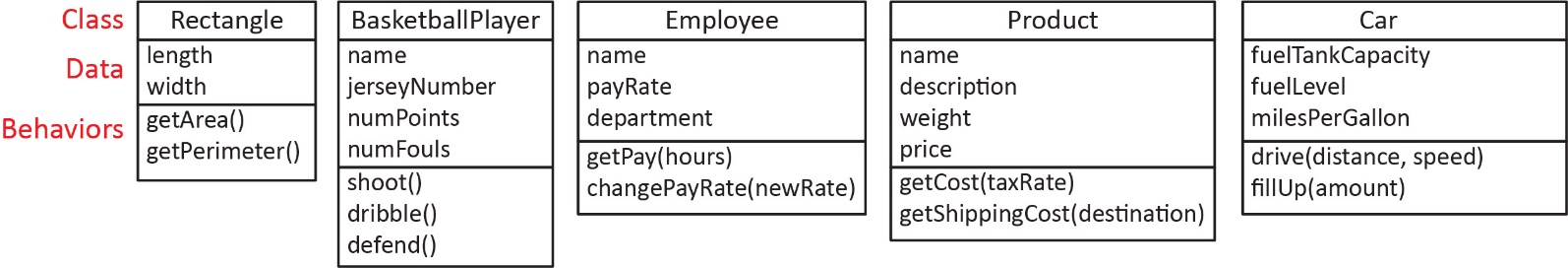
* Instance variables – information that the class stores.
* Methods – do something with or to the instance variables.
* Constructors – used to create an object from the class, giving the instance variables an initial value.

Collectively, we call these the *members* of a class. Each of the components is optional; however, most classes will have all three.

*Modelling* is the art of deciding what classes are needed to solve a problem, what the state and behavior of each is, and how the classes are associated with one another (this last aspect is considered in the next chapter). Modelling is a skill that takes a lot of practice. In this class I will almost always tell you (in English, or in a class diagram) what classes are needed and what the state and behaviors are and you will be responsible for writing the class.

Even the smallest of real systems will have 6 or more classes. Large systems could have hundreds. The group project for software engineering (CS 4321) typically has 10-25 classes.

Some simple examples of classes are shown below, whose state and behaviors should be self-explanatory. Of course, the actual state and behavior would depend on the context of the problem you are solving.



[Appendix 4](#Appendix_4) shows the state information for a network printer.

## Exercises

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer1* class)Write a class, *PiggyBank* to represent a piggy bank. This class should have instance variables to represent the number of quarters, dimes, and nickels (we will ignore pennies). This class also has a method, *getTotal* which returns the total amount of money in the bank. Also write code in main to create a piggy bank object, add some money (quarters, dimes, and nickels), and finally, print out the total in the bank with an informative message showing the total amount of money in the bank (calling the method to retrieve to total).
2. (Solution in *exercise\_cellular\_account* package, *CellularAccountVer1* class)Write a class, *CellularAccount* to represent an account a person would have for cell phone service. The class has these instance variables: (a) *minutesUsed* which is the number of minutes of cell service that has been used, (b) *costPerMin* which is the cost ($) per minute of cell phone service, (c) *minutesMax* which is the maximum number of minutes allowed for this account. The class also has a method, *getAmountDue* that returns the total charges for the account. All minutes used less than or equal to *minutesMax* are charged at the *costPerMin* rate. Any minutes over *minutesMax* are charged at a rate 40% higher than the base rate, *costPerMin*. For example:

Example 1: Suppose *minutesUsed* = 300.0, *minutesMax* = 500.0, *costPerMin*=0.05, then:

*getAmountDue()* = 300\*0.05 = $15.00

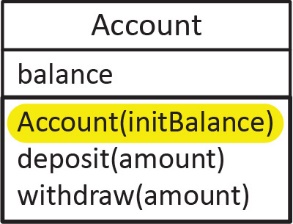
Example 2: Suppose *minutesUsed* = 600.0, *minutesMax* = 500.0, *costPerMin*=0.05, then:

*getAmountDue()* = 500\*0.05 + 100\*(1+0.4)\*0.05= $25.00 + $7.00 = $32.00

Finally, write code in *main* to implement the two examples above.

# Introduction to Constructors

For much of the remainder of this chapter, we will consider the *Account* class. As we learn new techniques, we will add to or change the class. The example below is found in *example\_account /Account2.java* in the code download.

Every class should have an explicit *constructor*. A constructoris like a method but it has no return type and it must have the same name as the class. It is used to create an object. Its job is to give the instance variables an initial value. 

For example, in the *Account* class below, we have added a constructor that defines a parameter, *initBalance* which represents the initial balance for the account when it is first created. The constructor takes this value and initializes the *balance* instance variable (it “remembers” the balance). We usually show the constructor at the top of the list of behaviors as shown in the class diagram above.

**public** **class** Account {

**public** **double** balance;

// Constructor

**public** Account(**double** initBalance) {

balance = initBalance;

}

**public** **void** deposit(**double** amount) {

**if**(amount>0) {

balance += amount;

}

}

**public** **void** withdraw(**double** amount) {

**if**(amount>0) {

balance -= amount;

}

}

}

Now, when we create an *Account* object (*account*), we can pass in an initial balance as show below.

**public** **static** **void** main(String[] args) {

Account account = **new** Account(1000.0);

account.deposit(200.0);

account.withdraw(400.0);

System.***out***.println("bal=" + account.balance); // 800.0

}

## Exercises

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer2* class)Add a constructor to the *PiggyBank* class from an earlier Exercise that has a constructor that accepts the number of quarters, dimes, and nickels that are initially in the bank. Also write code to test. You should write two snippets (small pieces of code) to do the following:
2. Create a *PiggyBank* object initialized with 3 quarters, 2 dimes, and 3 nickels. Print out the total in the bank with an informative message showing the total amount of money in the bank (calling the method to retrieve to total).
3. Create a *PiggyBank* object initialized with 3 quarters, 2 dimes, and 3 nickels. Then add another quarter, another dime, and a nickel. Print out the total in the bank with an informative message showing the total amount of money in the bank (calling the method to retrieve to total).
4. (Solution in *exercise\_cellular\_account* package, *CellularAccountVer2* class)Consider the *CellularAccount* class from an Exercise 2. Do the following:
5. Add a constructor that accepts *minutesMax* and *costPerMin*.
6. Write code in *main* to test the two examples from Exercise 2.

# Instance Variables & Getters

## *private* Instance Variables

In the example above, we declared the *balance* instance variable with *public* visibility:

**public** **double** balance;

This means that any other class in our system can see this variable, and, can access it directly and change it. For example, in *main* directly above, we used the *deposit* and *withdraw* methods to change the balance. However, we could have directly changed the balance as the code below shows:

**public** **class** AccountTest {

**public** **static** **void** main(String[] args) {

Account account = **new** Account(1000.0);

account.balance += 200.0;

account.balance -= 400.0;

System.***out***.println("bal=" + account.balance);

}

}

This violates an important design principle:

*Never let another class directly modify the value of an instance variable.*

Defining instance variables as *public* is poor practice and never done in industry. Best practice, is to declare all instance variables to be *private* (for the time being). For example:

**private** **double** balance;

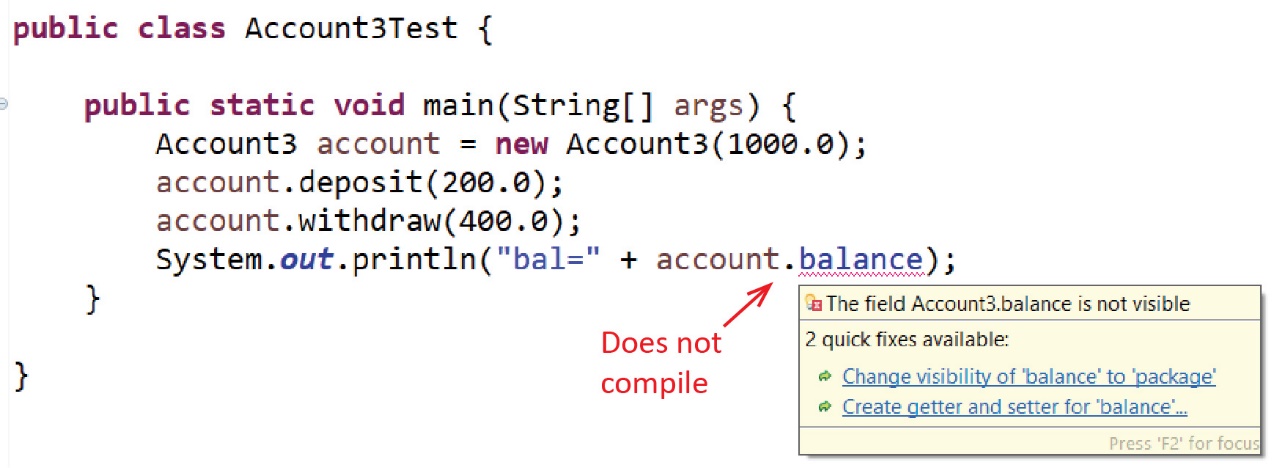
A *private* instance variable (or method) is only available (visible) inside the class itself. Thus, in the *deposit* method we wrote above, we can still access (and change) *balance* directly:

**public** **void** deposit(**double** amount) {

balance += amount;

}

However, other classes that use this *Account* class will not be able to access *balance* directly. For example, if *main* were defined in another class, *Account3Test*, as shown in the figure below, the reference to: account.balance will not compile.



From this point forward, all instance variables are declared as private. As we learn more visibility modifiers, this practice is modified to say that we declare instance variables with the least visibility as possible.

## Getters

If we make all instance variables private, then we are going to need a way to see what the value of the balance is from outside the class. To do this, we write a *public* method that simply returns the balance. Such a method is called a *getter* (also called an *accessor*) for the instance variable. A *getter,* in its simplest form, simply returns the value of the instance variable. The syntax is:

**public** DataTypeOfInstanceVariable getInstanceVariable() {

**return** instanceVariable;

}

For example:

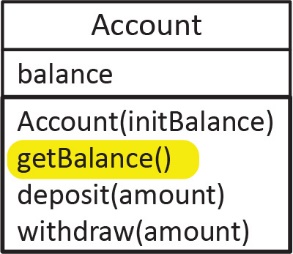
**public** **double** getBalance() {

**return** balance;

}

Note:

* A getter returns a copy (for primitive data types) of the instance variable. The true value is still inside the object. It does not allow the instance variable to be changed. We consider changing the value of an instance variable shortly, when we consider *setters*.
* The getter is declared *public* which means that any other class can use this method.
* Best practice is to name a getter: “get” + name of instance variable.

The class, with the addition of the getter, is shown below (see *example\_account/Account3.java* and *Account3Test* in the code download).

**public** **class** Account {

**private** **double** balance;

**public** Account(**double** initBalance) {

balance = initBalance;

}

**public** **double** getBalance() {

**return** balance;

}

**public** **void** deposit(**double** amount) {

**if**(amount>0) {

balance += amount;

}

}

**public** **void** withdraw(**double** amount) {

**if**(amount>0) {

balance -= amount;

}

}

}

Note that in *main*, we now use the getter, *getBalance* to access the value of *balance*

**public** **static** **void** main(String[] args) {

Account3 account = **new** Account3(1000.0);

account.deposit(200.0);

account.withdraw(400.0);

System.***out***.println("bal=" + account.getBalance());

// What does this code do?

**double** balance = account.getBalance();

balance = 1000000;

System.***out***.println("bal=" + account.getBalance());

}

This code displays:

bal=800.0

bal=800.0

The second-to-last and next-to-last lines (as repeated below) simply declare a local variable and assign it a large value. It does not change the balance that is inside the *account* object.

**double** balance = account.getBalance();

balance = 1000000;

Almost all instance variables will have a getter and most instance variables will be initialized through a parameter in the constructor.

**public** **class** Account {

**private** **double** balance;

**public** Account(**double** initBalance) {

balance = initBalance;

}

**public** **double** getBalance() {

**return** balance;

}

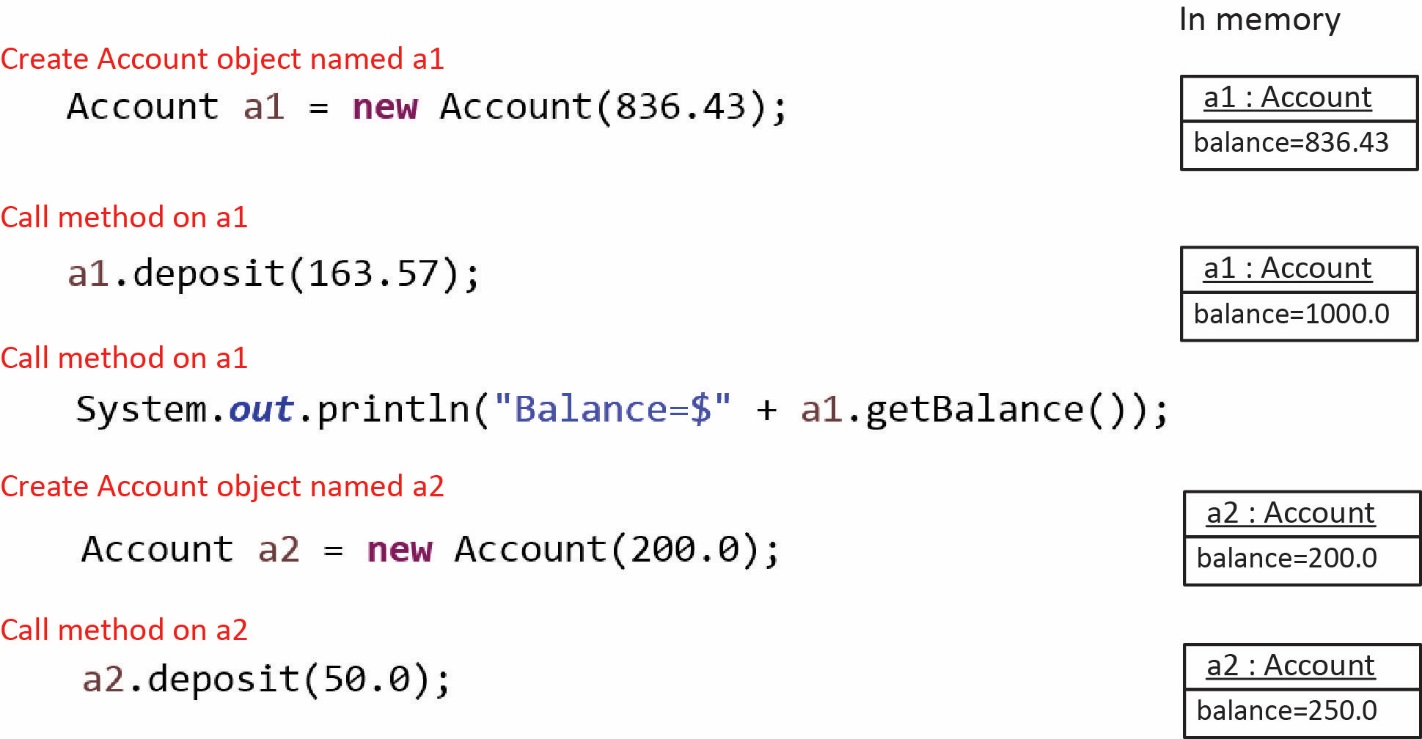
...

}

However, sometimes, it doesn’t make sense to require a parameter in the constructor to initialize an instance variable. For example, suppose we want the *Account* class to keep track of the number of withdrawals that have been made. Then we need to introduce an instance variable, *numWithdrawals* and a getter, *getNumWithdrawals*. The constructor doesn’t need to define a parameter for this instance variable. Instead, it can just set its value to 0. (see *example\_account/Account4.java* and *Account4Test* in the code download).

|  |  |
| --- | --- |
| *Account* Class | Test Code |
| **public** **class** Account {  **private** **double** balance;  **private** **int** numWithdrawals;    **public** Account(**double** initBalance) {  balance = initBalance;  numWithdrawals = 0;  }    **public** **int** getNumWithdrawals() {  **return** numWithdrawals;  }    **public** **void** withdraw(**double** amount) {  **if**(amount>0) {  balance -= amount;  numWithdrawals++;  }  }  ...  } | **public** **static** **void** main(String[] args) {  Account account = **new** Account(1000.0);  System.***out***.println(account.getNumWithdrawals()); // 0  account.deposit(200.0);  account.withdraw(400.0);  System.***out***.println(account.getNumWithdrawals()); // 1  account.withdraw(100.0);  System.***out***.println(account.getNumWithdrawals()); // 2  } |

We can create as many objects as we need in a program. For example, below, we create two *Account* objects, *a1* and *a2* which occupy separate places in memory. The diagrams on the right are called *object diagrams* and are used to show what is in memory as the program is running.



## An Array Instance Variable & Indexed Getter

Sometimes it is useful to have an instance variable that is an array. For example, suppose we are representing a professional basketball team and want to store the number of points the team scored in each quarter as well as the name of the team. We could store the number of points for each quarter in an array as shown below (see *example\_basketball\_array\_instance\_variable* packagein the code download):

**public** **class** BasketballTeam {

**private** String name;

**private** **int**[] points = **new** **int**[4];

**public** BasketballTeam(String name, **int** pointsQ1, **int** pointsQ2,

**int** pointsQ3, **int** pointsQ4) {

**this**.name = name;

points[0] = pointsQ1;

points[1] = pointsQ2;

points[2] = pointsQ3;

points[3] = pointsQ4;

}

}

If we want to provide a way to get the number of points for a particular quarter, then we need an *indexed getter*. For example:

**public** **int** getPoints(**int** quarter) {

**return** points[quarter];

}

It would probably be useful to validate the parameter, *quarter*, before using it as the only permissible values are: 0,1,2,3. Thus, we could write the getter as:

**public** **int** getPoints(**int** quarter) {

**if**( (quarter < 0) || (quarter>=points.length) ) {

**return** -1;

}

**return** points[quarter];

}

The *BasketballTeam* class might also have methods that compute values from the *points* array. For example:

1. Add a method to the *BasketballTeam* class, *totalPoints,* which returns the total number of points scored in the game.

**public** **int** totalPoints() {

**int** sum = 0;

**for**(**int** i=0; i<points.length; i++) {

sum += points[i];

}

**return** sum;

}

1. Add a method to the *BasketballTeam* class, *numQuartersMoreThan* that accepts an integer, *numPoints,* representing a number points. This method should return the number of quarters that the team score more than *numPoints.*

**public** **int** numQuartersMoreThan(**int** numPoints) {

**int** sum = 0;

**for**(**int** i=0; i<points.length; i++) {

**if**(points[i] > numPoints) {

sum++;

}

}

**return** sum;

}

Finally, we could use this class with code as shown below:

BasketballTeam team = **new** BasketballTeam("Hawks", 18, 23, 31, 26);

**int** q3Points = team.getPoints(2);

**int** totPoints = team.totalPoints();

**int** numQMore20 = team.numQuartersMoreThan(20);

Finally, note: **None of the instance variables nor methods are *static*. We will almost never use the *static* modifier in classes we write.** However, *main* will always be static. We discuss the meaning of *static* a later [section](#_Static_Variables_&).

# Setters

If we want to provide a way to change the value of a private instance variable, we write a method called a *setter* (also called a *mutator*). A setter, is a method that replaces the value of an instance variable with a new value that is supplied as an argument to the setter. The syntax is:

**public** **void** setInstanceVariable(DataType newValue) {

instanceVariable = newValue;

}

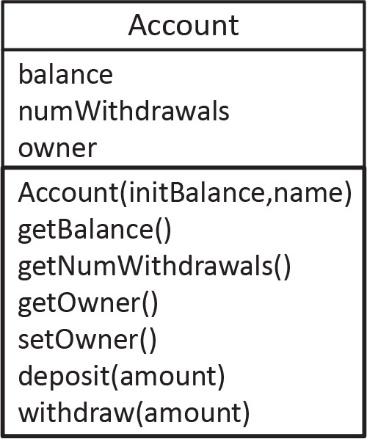
The convention for naming a setter is: “set” + the name of the instance variable.

For example, in the *Account* class, let’s introduce an instance variable for the name of the owner of the account.

**private** String owner;

Then, define a getter as usual, and also define a setter so that the name can be changed:

|  |  |
| --- | --- |
| **Getter** | **Setter** |
| **public** String getOwner() {  **return** owner;  } | **public** **void** setOwner(String newOwner) {  owner = newOwner;  } |

A portion of the new class is shown below. Note that we have also added the name to the constructor (see *example\_account/Account5.java* and *Account5Test* in code download).

**public** **class** Account {

**private** **double** balance;

**private** **int** numWithdrawals;

**private** String owner;

**public** Account(**double** initBalance, String name) {

balance = initBalance;

owner = name;

numWithdrawals = 0;

}

**public** String getOwner() {

**return** owner;

}

**public** **void** setOwner(String name) {

owner = name;

}

...

}

We could use the class above, with code like this:

**public** **static** **void** main(String[] args) {

Account a = **new** Account(500.0, "Xavier");

System.***out***.println("Name=" + a.getOwner()); // Xavier

a.setOwner("Marco");

System.***out***.println("Name=" + a.getOwner()); // Marco

}

You may ask why we didn’t define a setter for the *balance*. There are two main reasons. First, we don’t need one. The *withdraw* and *deposit* methods allow the balance to be changed. Second, we probably don’t want to allow the *balance* to be changed except through these two methods (and perhaps others, like: *applyInterest*). In other words, we don’t want to let clients (other code) to just set the balance to some arbitrary value, *e.g. account.setBalance(1000000.0)*. This is called *defensive design.*

## Indexed Setter

Consider the *BasketballTeam* class from a previous [section](#_An_Array_Instance) where we had a *points* instance variable that was an array of integers representing the number of points the team scored in each of 4 quarters. If we want to provide a way to change the number of points for a particular quarter, then we need an *indexed setter* that accepts the quarter to change, and new number of points. For example (see *example\_basketball\_array\_instance\_variable* packagein the code download):

**public** **void** setPoints(**int** quarter, **int** numPoints) {

**if**( (quarter < 0) || (quarter>=points.length) ) {

**return**;

}

points[quarter] = numPoints;

}

Finally, we could use this class with code as shown below:

BasketballTeam team = **new** BasketballTeam("Hawks", 18, 23, 31, 26);

team.setPoints(3,33); // Change 4th quarter points

## Exercises

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer3* class)Change the *PiggyBank* class from an earlier Exercise so that:

* All instance variables are private.
* All instance variables have a getter.
* All instance variables have a setter.

1. Write code in main to test these new methods in the following way:

* Create a *PiggyBank* object initialized with 3 quarters, 2 dimes, and 3 nickels.
* Then, add 2 quarters. You could do this with this code:

PiggyBank pBank = new PiggyBank(3,2,3);

pBank.setNumQuarters(5);

However, this is not a good solution because we are using knowledge that is in our head of how many quarters are already in the bank. It is much better to assume we don’t know how many quarters there are in the bank. Remember, we were asked to “add 2 quarters”. Thus, a better solution is:

PiggyBank pBank = new PiggyBank(3,2,3);

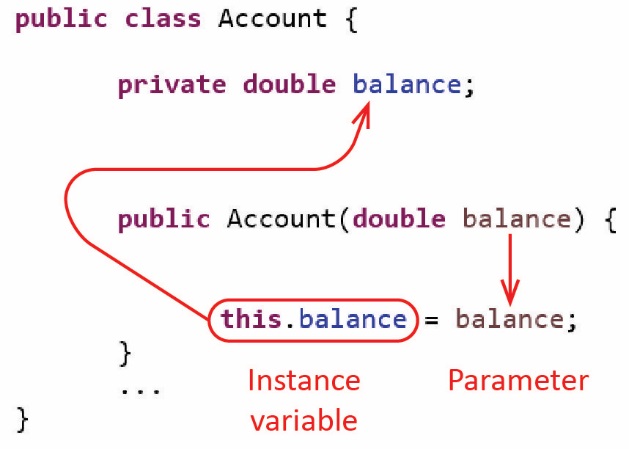
int numQuarters = pBank.getNumQuarters() + 2;

pBank.setNumQuarters(numQuarters);

* Add 3 nickels
* Print out the total in the bank with an informative message showing the total amount of money in the bank (calling the method to retrieve to total).

1. (Solution in *exercise\_cellular\_account* package, *CellularAccountVer3* class)Consider the *CellularAccount* class from an Exercise 2. Do the following:
2. All instance variables are private.
3. The *minutesMax* and *costPerMin*  instance variables have getters and setters.
4. The *minutesUsed* instance variable has a getter, but no setter.
5. Add a method, *makeCall* that accepts the length (minutes) of a call. This method will simply increase *minutesUsed* by this amount. In other words, instead of having a setter for *minutesUsed,* we have a more intuitive method which will handle the update to *minutesUsed*.
6. Write code in main to test these new methods thoroughly.

# The *this* Reference

The *this* keyword can be used inside a class to refer to itself (kind of like saying, “me” or “my” when speaking about yourself). For now, the only use of *this* that we consider is that it is a way to differentiate between an instance variable and a parameter with the same name. Frequently, we name the constructor parameters using the same name as the corresponding instance variables as shown on the right.

Thus, *this.balance* refers to the instance variable while *balance* is a parameter, which means it is a local variable.

A mistake is shown below. The code has no affect: it simply assigns *balance* to *balance.* The code does compile; however, it is incorrect as it does not initialize the instance variable. The parameter, *balance* is a local variable, thus, it hides the instance variable (unless we use *this* as shown above):

**public** Account(**double** balance) {

balance = balance;

}

## Exercises

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer4* class)Change the *PiggyBank* class from an earlier Exercise so that all the parameters in the constructor have exactly the same name as the corresponding instance variables.
2. (Solution in *exercise\_cellular\_account* package, *CellularAccountVer4* class)Change the *CellularAccount* class from an earlier Exercise so that all the parameters in the constructor have exactly the same name as the corresponding instance variables.

# The *toString* Method

Every class should have a *toString* method. The signature must be:

**public** String toString()

The *toString* method should return a string representation of the object. In other words, a descriptive string showing the values of the instance variables. If the class has methods that take no arguments and return a value, they should be executed, and the result displayed, in *toString* also. Displaying the result of the *toString* method is useful for debugging and testing. For example, for the *Account* class, the *toString* method is (see *example\_account/Account6.java* and *Account6Test* in code download):

@Override

**public** String toString() {

String msg = "owner=" + owner + ", bal=" + balance +

", num withdrawals=" + numWithdrawals;

**return** msg;

}

Technically, we are *overriding* the *toString* method which we will discuss in a later chapter. The “@Override” is called an *annotation*. It is not required, but is a *best practice*.

You can explicitly call the *toString* method:

Account a = **new** Account(500.0, "Xavier");

a.deposit(1000.0);

a.withdraw(300.0);

System.***out***.println(a.toString()); // owner=Xavier, bal=1200.0, num withdrawals=1

However, you don’t have to explicitly call *toString*. If you simply “print the object”, Java implicitly calls the *toString* method. For example, these two lines are identical

System.***out***.println(a);

System.***out***.println(a.toString());

Sometimes it is useful to include the class name in the *toString* method so that when you are looking at output you know which class the message originated from. This might be particularly useful, for example if you had several different account classes: *CheckingAccount*, *SavingsAccount, etc*., each with a *balance*. For example:

@Override

**public** String toString() {

**return** "CheckingAccount: balance=" + balance;

}

If you do not write your own *toString* method, there still is an *inherited* one (we discuss this in a later chapter). This inherited one returns a string that consists of:

ClassName@hexadecimalNumber

For example:

Account@15db9742

The only reason we mention this is that you are likely to come across this at some point when you print an object. Thus, when you do, **you will immediately recognize that you forgot to write (override) the *toString* method.**

## Exercises

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer5* class) Change the *PiggyBank* class from an earlier Exercise by adding a *toString* method that returns an informative message detailing how many of each coin are present and the total. For example, if this code is executed:

PiggyBank pBank = new PiggyBank(3,2,3);

int numQuarters = pBank.getNumQuarters() + 2;

pBank.setNumQuarters(numQuarters);

System.out.println(pBank);

The output should look similar to this:

Quarters=5, Dimes=2, Nickels=3, Total=$1.60

Note, the total may have a lot of decimals, *e.g.* 1.600000000001, or something like this. This is ok. We will learn to format out output later.

1. (Solution in *exercise\_cellular\_account* package, *CellularAccountVer5* class)Change the *CellularAccount* class from an earlier Exercise by adding a *toString* method that returns an informative message similar to what is shown below.

Amount Due=$32.00, Used=600.0 min., Max=500.0 min., Cost Per Min=0.05

# Testing a Class

|  |
| --- |
| ***Account* Class** |
| **public** **class** Account {  **private** **double** balance;    **public** Account(**double** initialBalance) {  balance = initialBalance;  }  ...  **public** **static** **void** main(String[] args) {  Account a1 = **new** Account(836.43);  a1.deposit(163.57);  System.***out***.println("Balance=$" +  a1.getBalance());  ...  }  } |

When writing classes, we usually do *informal testing*, where we put a *main* in the class we are writing as shown in the example on the right. There, we just write simple code to illustrate how the class works, how to use the methods, *etc.*

However, all classes must be tested formally. For *formal testing*:

* We write a separate test class to test each class. The naming convention of this test class is: ClassNameTest. For example, if the class is *Test,* then the test class is *AccountTest*.
* Write at least one static test method for every method in the class, except those with no logic. In other words, we don’t test methods unless they do some computation, or have a condition (an *if* statement). Thus, many times getters and setters are not tested as the only thing they do is return a value, or set a value. Also, constructors are not usually tested for the same reason. However, as a beginner, I do find it useful for you to test constructors as mistakes can be caught quickly.
* The static test methods are called from main.
* Each test method has a name, prefaced by “test” and followed by a description of the test. For example, for the *deposit* method, we might have a test method, *testDeposit\_NegativeAmount* where we test the *deposit* method by passing a negative value to it.
* Each test is *stand-alone*. This means that everything that is needed for the test is created in the test method itself.
* The tests display the *expected output* and the *actual output*.
* The test methods are usually ordered so that the simplest ones are earlier. For example, if you need to test method *y,* but in doing so, you need to use method *x,* the *testX* method should be executed before the *testY* method.
* Test each method under all relevant conditions. For example, the *deposit* method we have considered:
  + It takes two different courses of action, depending on whether the amount being deposited is positive or negative. Thus, there would be two test methods for these two cases: *testDeposit\_PositiveAmount* and *testDepositNegativeAmount*.
  + Every time it is called, it increments the balance. Thus, we should test it after a single deposit. Then, we should have another test method to test the result when several successive deposits are made. Thus, we might have third test method: *testDeposit\_MultipleDeposits*

A complete set of test cases for the most recent version of the *Account* class is shown below (see *example\_account/Account6Test.java* in code download)

**public** **class** Account6Test {

**public** **static** **void** main(String[] args) {

*testAccountCreation*();

*testDeposit*();

*testDeposit\_NegativeAmount*();

*testDeposit\_Multiple*();

*testWithdraw*();

*testWithdraw\_NegativeAmount*();

*testWithdraw\_Multiple*();

*testDeposit\_Withdraw*();

}

**private** **static** **void** testAccountCreation() {

System.***out***.println("-->testAccountCreation()");

Account6 a1 = **new** Account6(1000.0, "Dalton");

String expected = "Expected: owner=Dalton, bal=1000.0, num withdrawals=0";

String actual = " Actual: " + a1.toString();

System.***out***.println(expected);

System.***out***.println(actual);

}

**private** **static** **void** testDeposit() {

System.***out***.println("\n-->testDeposit()");

Account6 a1 = **new** Account6(1000.0, "Leah");

a1.deposit(500.0);

System.***out***.println("balance should be $1500, balance=$" + a1.getBalance());

}

**private** **static** **void** testDeposit\_NegativeAmount() {

System.***out***.println("\n-->testDeposit\_NegativeAmount()");

Account6 a1 = **new** Account6(1000.0, "Leah");

a1.deposit(-500.0);

System.***out***.println("balance should be $1000, balance=$" + a1.getBalance());

}

**private** **static** **void** testDeposit\_Multiple() {

System.***out***.println("\n-->testDeposit\_Multiple()");

Account6 a1 = **new** Account6(1000.0, "Avery");

a1.deposit(500.0);

a1.deposit(300.0);

System.***out***.println("balance should be $1800, balance=$" + a1.getBalance());

}

**private** **static** **void** testWithdraw() {

System.***out***.println("\n-->testWithdraw()");

Account6 a1 = **new** Account6(1000.0, "Leah");

a1.withdraw(500.0);

System.***out***.println("balance should be $500, balance=$" + a1.getBalance());

System.***out***.println("num withdrawals should be 1, numWithdrawals=" + a1.getNumWithdrawals());

}

**private** **static** **void** testWithdraw\_NegativeAmount() {

System.***out***.println("\n-->testDeposit\_NegativeAmount()");

Account6 a1 = **new** Account6(1000.0, "Leah");

a1.withdraw(-500.0);

System.***out***.println("balance should be $1000, balance=$" + a1.getBalance());

System.***out***.println("num withdrawals should be 0, numWithdrawals=" + a1.getNumWithdrawals());

}

**private** **static** **void** testWithdraw\_Multiple() {

System.***out***.println("\n-->testWithdraw\_Multiple()");

Account6 a1 = **new** Account6(1000.0, "Avery");

a1.withdraw(200.0);

a1.withdraw(300.0);

System.***out***.println("balance should be $500, balance=$" + a1.getBalance());

System.***out***.println("num withdrawals should be 2, numWithdrawals=" + a1.getNumWithdrawals());

}

**private** **static** **void** testDeposit\_Withdraw() {

System.***out***.println("\n-->testDeposit\_Withdraw()");

Account6 a1 = **new** Account6(1000.0, "Zoe");

a1.withdraw(500.0);

a1.deposit(300.0);

a1.deposit(400.0);

System.***out***.println("balance should be $1200, balance=$" + a1.getBalance());

System.***out***.println("num withdrawals should be 1, numWithdrawals=" + a1.getNumWithdrawals());

}

}

Summary: so far, we have talked about writing a class and testing a class. We have not illustrated using a class to solve a problem. For example, a problem might be: write an app that allows a user to maintain and manipulate a bank account. So, the first step would be to write and test an *Account* class as we have done so far. After that, we would write an app that utilizes this class. For example, we might have a menu (or Gui) where a person can create an account by suppling their name and initial balance. Then, other options on the menu would be to enter an amount to deposit or withdraw, *etc*. We won’t consider writing apps until later when we consider Gui applications.

## Exercises

1. (Solution in *exercise\_savings\_account* package) Write a class for the situation where we need to represent a savings account. A *SavingsAccount:*

* Has an owner (name) and a balance (hint: these are instance variables), which are supplied when the account is first created. (hint: this is referring to the constructor)
* The owner and balance should not be changed directly. Hint: define getters, but not setters
* A *deposit* method that accepts an amount of money, which must be positive. If it is positive, then it is added to the balance.
* A *withdraw* method that accepts an amount of money, which must be positive. If it is positive, then it is deducted from the balance. If the balance goes below 0.0, then a service fee of $20 is charged (reduces the balance by this amount). For example, if the balance is $100, and a withdrawal of $120 is made, then the balance becomes: $100-$120-$20 (service fee) = -$40.
* The total of all services fee should be available. Hint: (a) define a *serviceFeeTotal* instance variable and initialize it to 0.0, (b) define a getter for it, (c) Update the withdraw method to increase the *serviceFeeTotal* when appropriate.
* An *applyInterest* method accepts an interest rate as a decimal argument (*e.g.* 0.05 for 5%). This method increases the balance by this percent, provided the balance is greater than 0. For example, if the balance is $100 and *applyInterest(0.05)* is called, then the balance becomes: $100\*(1+0.05) = $105.
* Write an *SavingsAccountTest* class to test all the methods.

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer6* class) Change the *PiggyBank* class from an earlier Exercise by doing the following:
2. Remove all the setters and replace them with *add* methods: *addQuarters(numQuarters:int), addDimes(numDimes:int), addNickels(numNickels:int)*. For example, the *addQuarters* method should just add the parameter to the instance variable. However, it should only do this if the parameter is positive; otherwise, it should do nothing. The other add methods are similar.

Note: the setters we added earlier were not a good design choice. We simply did it to get some practice writing setters. The reason it is a bad design choice is that we should write methods that mimic how we would use such an object in the real world. For example, two add 2 quarters to the bank, it is more intuitive to write a statement like this:

PiggyBank pBank = new PiggyBank(3,2,3);

pBank.addQuarters(2);

System.out.println(pBank);

1. Write a method, *withdrawQuarters(numQuarters:int)* that removes this number of quarters from the bank, if possible. For example:

PiggyBank pBank = new PiggyBank(7,2,3);

pBank.withdrawQuarters(5);

System.out.println(pBank);

Would produce this output:

Quarters=2, Dimes=1, Nickels=3, Total=$0.85

Another example:

PiggyBank pBank = new PiggyBank(7,2,3);

pBank.withdrawQuarters(8);

System.out.println(pBank);

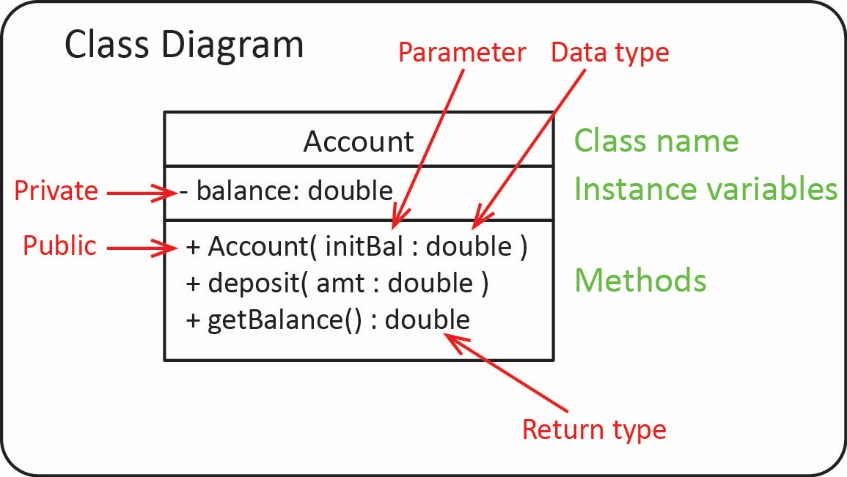
Would produce this output:

Quarters=7, Dimes=1, Nickels=3, Total=$2.00

1. Write a *PiggyBankTest* class to test all the methods. Remember: (a) you don’t need to test the getters as there is no logic in the code, (b) you will have two test methods for each of the other methods.
2. (Solution in *exercise\_cellular\_account* package, *CellularAccountVer6* class)Change the *CellularAccount* class from an earlier Exercise.Write a *CellularAccountTest* class, with test methods to thoroughly test all the methods.

# Class & Object Diagrams

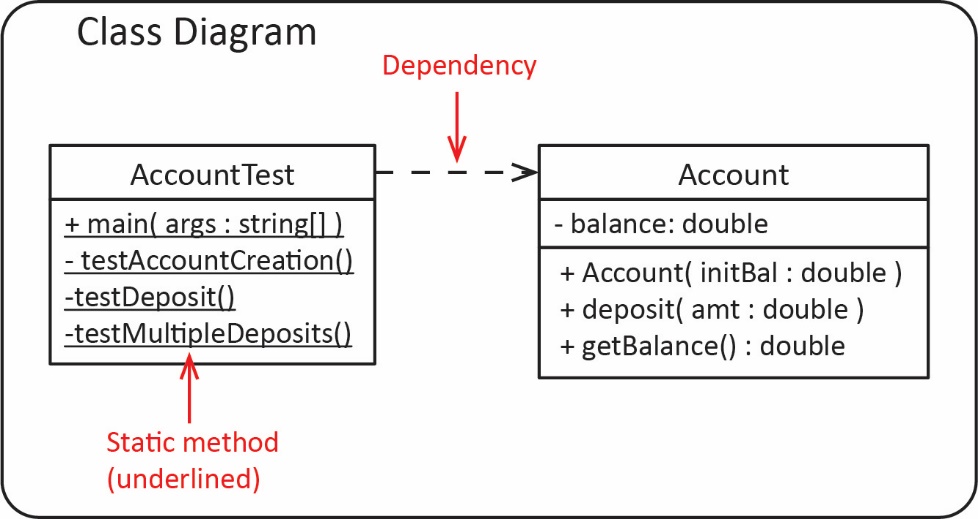
The Unified Modelling Language (UML) is a standardized graphical language for modelling object-oriented software. UML is composed of a number of different diagrams, one of which is the *class diagram*.

A UML *class diagram* is used to model the classes in a system. An example is shown on the right. Note the following:

* A class diagram has three compartments: the top box shows the name of the class, the middle box shows the instance variables, and the bottom box shows the methods and constructors.
* The visibility is specified for each member where we use “-“ for private and “+” for public.
* The notation is a little bit different than Java. This is because UML for modelling object-oriented systems that can be developed in any object-oriented language. For example, the syntax for:

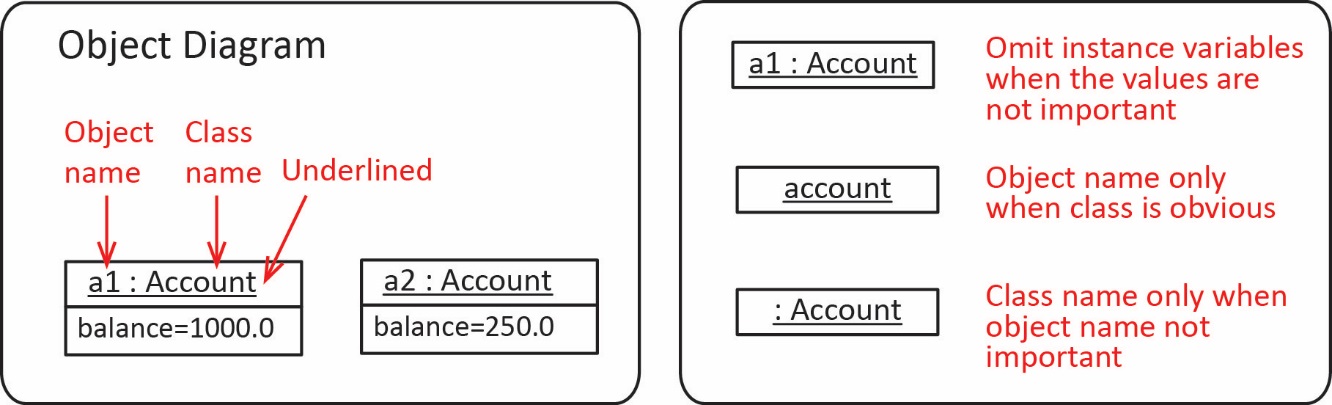
**Instance Variables:** varName : datatype

**Methods:** method( paramName : dataType ) : returnType

****We can show the test class in a class diagram. Note the following:

* Static methods are underlined
* We indicate a *dependency* relationship between the two classes which is depicted with a dashed line with an arrow pointing to the dependency. In this case, the *AccountTester* class depends on the *Account* class (because the test methods use the *Account* class).

An *object diagram* shows an example of *objects* (instances of a class) that might exist in a system when it is running. In other words, it shows what is in memory at some point during program execution. As we consider more complex designs, with relationships between classes, *etc.,* these become more useful. The syntax for elements in an *object diagram* is show by way of example in the figures below.

****

# Best Practices – Naming Conventions

1. Class names should be nouns, in mixed case with the first letter of each word capitalized (this is referred to as *camel case*), and almost always singular. Examples:

BankAccount, Employee, BasketballPlayer

Class names should be simple, but descriptive and use whole words, not acronyms or abbreviations (unless they are widely used like: SSN, URL, HTML, *etc.*)

1. Method names should be verbs or verb phrases and use camel case with the first letter of the first word in lowercase. They should not be too long, but long enough to make it clear the purpose of the method. Examples:

getBalance(), initializeGame(), calculateTax(), buildAccountReport()

1. Variable names should be short, but descriptive, and use camel case with the first letter of the first word in lowercase. Examples:

numWithdrawals, shotsAttempted, shotsMade

1. A boolean variableshould usually begin with “is”. For example:

**boolean** isPasswordValid, isInitialized, isGameOver;

Sometimes these prefixes are more suitable: “has”, “can”, “should”. For example:

**boolean** hasMetRequirements, canTurnLeft, shouldAuthenticate;

1. Suppose you have a boolean instance variable:

**private** **boolean** isComplete;

then the naming convention for the corresponding getter and setter are:

**public** **boolean** isComplete() {

**return** isComplete;

}

**public** **void** setComplete(**boolean** isComplete) {

**this**.isComplete = isComplete;

}

1. Example – (Solution in *example\_basketball\_player\_fouls* package) Model the number of fouls a basketball player has and if they are fouled out (5 fouls).

|  |  |
| --- | --- |
| *BasketballPlayer* Class | *BasketballPlayer* Class (continued) |
| **public** **class** BasketballPlayer {  ...  **private** **int** numFouls=0;  **private** **boolean** isFouledOut=**false**;    ...  **public** **boolean** isFouledOut() {  **return** isFouledOut;  }  **public** **int** getNumFouls() {  **return** numFouls;  } | **public** **void** commitFoul() {  **if**(!isFouledOut) {  numFouls++;  **if**(numFouls==5) {  isFouledOut = **true**;  }  }  }  ...  } |

1. A boolean method (or expression) returns *true* or *false.* Thus, in an *if* statement you should not explicitly check to see if it is *true* (or *false*). For example:

|  |  |  |
| --- | --- | --- |
| Best Practice |  | Not Preferred |
| **if**(isFouledOut)) |  | **if**(isFouledOut==**true**) |
| **if**(!isFouledOut) |  | **if**(isFouledOut==**false**) |

## Exercises

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer7* class) Change the *PiggyBank* class from an earlier Exercise by doing the following:
2. Add an instance variable, *isEmpty* that is *true* if the total in the bank is 0.0 and *false* otherwise.
3. Add a getter for this instance variable (use best practice for naming the getter).
4. Add some test methods to *PiggyBankTest* to test this.
5. (Solution in *exercise\_cellular\_account* package, *CellularAccountVer7* class)Change the *CellularAccount* class from an earlier Exercise by doing the following:
6. Add an instance variable, *isNearMax* that is *true* if the total minutes used is within 50 minutes of the maximum. Otherwise, it should return false.
7. Add a getter for this instance variable (use best practice for naming the getter).
8. Add some test methods to *CellularAccountTest* to test this.

# Best Practices – Writing Classes

1. A *domain (or custom) class* is a class that models a real-world entity such as we have discussed so far, *e.g.* *Employee, Account, Bank, etc.* A domain class should never print anything. It is OK to put print statements in while you are debugging; however, they must be removed or commented out when debugging is complete.

Consider the *bark* method in the *Dog* class on the left below where it prints a message to the consule. This is not preferred. We prefer the *bark* method on the right that returns a string. This provides more flexibility as the calling code can then decide what to do with it, *e.g.* display it on the console (as shown in *main* below), display it in a Gui, store it in a file or database, send it as a message to some other entity, *etc.*

|  |  |
| --- | --- |
| Not preferred | Preferred (Required!) |
| **public** **class** Dog {  **private** String name;  **public** Dog(String name) {  **this**.name = name;  }    **public** **void** bark() {  System.***out***.println(name + " barks");  }  **public** **static** **void** main(String[] args) {  Dog d = **new** Dog("Zoro");  d.bark();  }  } | **public** **class** Dog {  **private** String name;  **public** Dog(String name) {  **this**.name = name;  }    **public** String bark() {  **return** name + " barks";  }  **public** **static** **void** main(String[] args){  Dog d = **new** Dog("Zoro");  System.***out***.println(d.bark());  }  } |

Test classes will, of course, have print statements.

1. Any code inside a block (an open/close pair of braces) should be consistently indented. Generally, Eclipse (or any IDE) will take care of this for you. See [Curly brackets and indentation](https://courses.cs.washington.edu/courses/cse143/16au/style/curly-brackets-and-indentation.html), and [Formatting lines](https://courses.cs.washington.edu/courses/cse143/16au/style/formatting-lines.html).
2. *if, else, else if* should always be followed by a set of braces, even if there is only one statement. For example:

|  |  |
| --- | --- |
| Not preferred | Preferred |
| **if**(weight < 100.0)  numLight++;  **else**  numHeavy++; | **if**(weight < 100.0) {  numLight++;  }  **else** {  numHeavy++;  } |

1. The members of a class should be ordered in a consistent fashion. The order below is somewhat standard:

**public** **class** MemberOrder {

// Instance variables

// Constructors

// Getters & Setters (in pairs)

// Methods (alphabetic)

// toString

// main

}

1. Consistent indenting and braces with conditionals and loops is required and would not be allowed into real-world production code if there are violations. Consider Google’s [Java Style Guide](https://google.github.io/styleguide/javaguide.html). Many companies use a [*linter*](https://en.wikipedia.org/wiki/Lint_(software))to analyze their code to check for violations of these and other style requirements.

# The String.format Method

The code for the examples that follow are found in the *example\_string\_format* package.

The *String* class, which was covered in CS 1301, will be used often in homework and test problems. You should be familiar with these methods in the *String* and *Character* classes. If needed, [Appendix 1](#Appendix_1) provides a review of these classes.

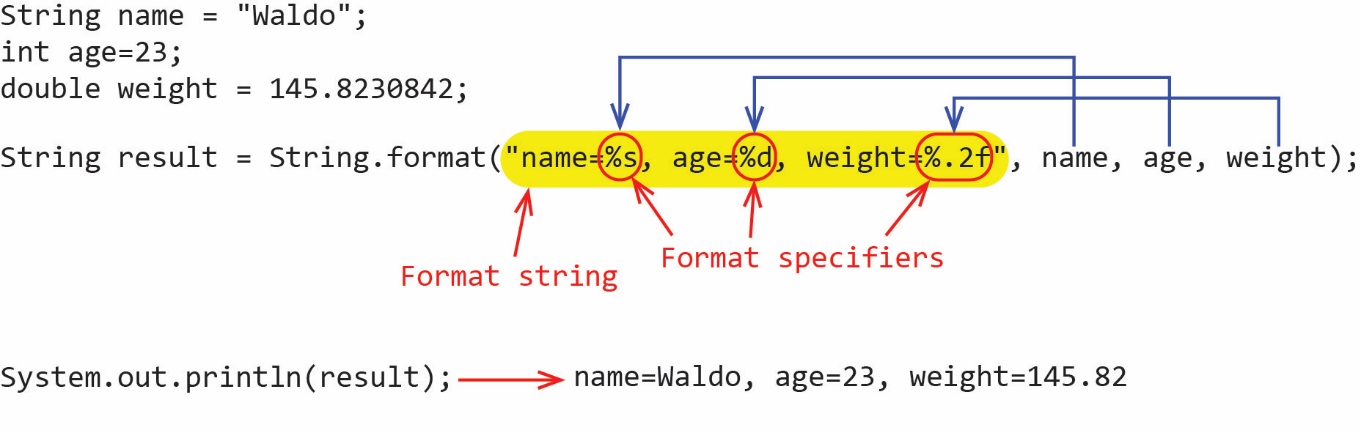
* *Character* Class – static methods: *isDigit(char), isLetter(char), isLetterOrDigit(char), isLowerCase(char), isUpperCase(char), toLowerCase(char), toUpperCase(char)*
* *String* Class – *length, charAt(i), equals, substring(i,j), substring(i), indexOf(ch/str), indexOf(ch/str,i), lastIndexOf(ch/str), lastIndexOf(ch/str,i), toLowerCase(), toUpperCase()*

In this section, we consider the *String* class’s static method, *format,* which you probably did not cover in CS 1301. It is a simple way to make output look nice. It can be tedious at first, but it is a must-have skill: you don’t want sloppy output in a system.

The syntax for the *format* method is:

String result = String.format( “format string” [, arg1, arg2, … ] );

The *format string* is composed of *string literals* and *format specifiers*. Format specifiers define how to format the arguments. For example:



Note:

* There is one format string (yellow highlight above).
* There is one argument for each format specifier. The order of the arguments matters. In other words, the first argument is substituted into the first format specifier, *etc.*
* The methodreturns a string with the arguments substituted for the format specifiers
* A “%” is used to signal the beginning of a format specifier.
* In the example above, there are 3 format specifiers:

1. “%s” – The corresponding argument is a string or character. No special formatting is done, the argument is simply substituted directly. Additional values can be used. For example: “%20s” makes the field width 20 characters, and the string is right justified by default. Similarly, “%-20s” would left justify.
2. “%d” – The corresponding argument is an integer (or byte, short, long). As with a string format specifier, justification and field with can be specified.
3. “%.2f” – The corresponding argument is a floating-point number, either float or double.

A line-break can be inserted by adding “\n”, or the newline format specifier, “%n”:

String result2 = String.*format*("weight=%,.2f%n", weight);

String result3 = String.*format*("weight=%,.2f\n", weight);

The format specifier for a floating-point number can also take a comma to indicate to use a comma as the thousand’s separator:

% *, . decimals f*

For example:

|  |  |
| --- | --- |
| **Code** | **Output** |
| **double** salary = 78224.8230842;  result = String.*format*("Salary=$%,.2f", salary); | Salary=$78,224.82 |

Note that the “$” is not a part of the format specifier, it is a string literal.

The format specifier for a floating-point number can also take the width (number of columns) of the space to place the number:

% *width . decimals f*

As an example, the code below specifies a field width of 8 with 2 decimal places:

|  |  |
| --- | --- |
| **Code** | **Output** |
| **double** x = 498.57334;  result = String.*format*("%8.2f", x);  System.***out***.println(result); | G:\eDataClasses\CS 1302 - Programming 2\notes\01_ch4_Strings\a3.jpg |

Notes:

* You can also use a “-“ flag in front of the width to left-justify
* If the specified field width is not wide enough, The JVM just expands it so that the whole number is displayed.

Note that the *format string* is a *String*. This means that we can build it programmatically. For example, we could write a method that accepts a number, and a number of decimals and returns the number, as a string, with that number of decimals:

**public** String formatNum(**double** num, **int** numDecimals) {

String formatSpecifier = "%,." + numDecimals + "f";

String formattedNum = String.*format*(formatSpecifier, num);

**return** formattedNum;

}

Note that if there is an error in a format specifier, then a runtime error will result. For example, either of the two examples below result in a runtime error:

|  |  |  |
| --- | --- | --- |
| “%.2z” is not a valid format specifier |  | “%.2f” is not a valid format specifier for a string argument |
| **double** x = 498.57334;  result = String.*format*("x=%.2z", x);  System.***out***.println(result); |  | String val = "cat";  result = String.*format*("val=%.2f", val);  System.***out***.println(result); |

Finally, a [quick reference](https://www.cs.colostate.edu/~cs160/.Summer16/resources/Java_printf_method_quick_reference.pdf)[[2]](#footnote-2) for *String.format*.

## Exercises

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer8* class) Change the *toString* method in the *PiggyBank* class from an earlier Exercise so that it uses *String.format* to format the return string such that exactly 2 decimals are shown, and uses a “,” for the 1000’s separator. Write some informal test code in a *main* to test this method. Hint: you will need more than 4000 quarters to have a result more than $1000.
2. (Solution in *exercise\_build\_report* package) Write a static method, *buildReport* that accepts: an array of strings, names; an array of ints, ages; and an array of doubles, salaries. The size of each array is the same. The method should use *String.format* to build and return a string that shows each name, age, and salary on a single line in a format as shown below. Write some informal test code.

Name: Jed, age:22, salary: $48,339.23

Name: Keisha, age:33, salary: $68,992.92

Name: Jaylen, age:44, salary: $121,042.04

1. (Solution in *exercise\_cellular\_account* package, *CellularAccountVer7* class)Change the *toString* method in the *CellularAccount* class from an earlier Exercise so that it uses *String.format* to format the return string such as shown below. In other words, “$” and 2 decimals for the amount due, one decimal for minutes used and max mainutes, and 2 decimals for the cost per minute.

Amount Due=$32.00, Used=600.0 min., Max=500.0 min., Cost Per Min=0.05

# Helper Methods

Methods should be readable and as short as possible. If a method is long, it usually should be broken down into pieces of related functionality which are implemented by *private helper methods*. For example, suppose we have a method with two longish blocks of code, where each block logically performs some task:

**public** **void** longMethod() {

// a few or more lines of code to do task A

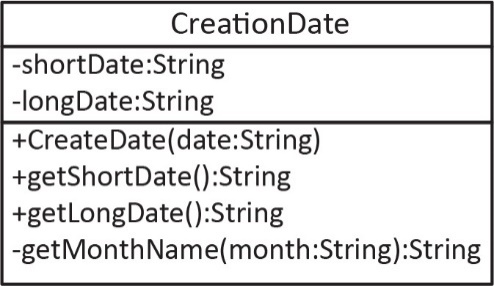
// a few or more lines of code to do task B

}

A better way to implement this method is to use helper methods:

|  |  |
| --- | --- |
| Long Method | Helper Methods |
| **public** **void** longMethod() {  taskA();  taskB();  } | **private** **void** taskA() {  // a few or more lines of code to do task A  }  **private** **void** taskB() {  // a few or more lines of code to do task B  } |

Helper methods generally have *private* visibility because we don’t necessarily want code outside the class to call them. And, in fact, there could be harm if an outside class, for example, called a helper method in the wrong situation, and/or didn’t call several of them in a particular sequence. As stated earlier, this is called *defensive coding*.

Example – Write a class, *CreationDate* that accepts a string in the format: *mmddyyyy* which represents a date. This class should have two methods which display the date in different formats. The *getShortDate* method should return a string formatted like this: mm/dd/yyyy. The *getLongDate* method should return a string formatted using a format like this: January 12, 2019. For example:

CreationDate cd1 = **new** CreationDate("04052019");

System.***out***.println(cd1.getShortDate()); // 04/05/2019

System.***out***.println(cd1.getLongDate()); // April 05, 2019

The solution is found in the *­example\_creation\_date* package. In the solution below, the constructor uses a helper method, *getMonthName* to convert the string version of the month (*e.g.* “04”) to the corresponding name for the month (*e.g.* April):

**public** **class** CreationDate {

**private** String shortDate;

**private** String longDate;

**public** CreationDate(String date) {

String month = date.substring(0,2);

String monthName = getMonthName(month); // Helper method

String day = date.substring(2,4);

String year = date.substring(4);

**this**.shortDate = month + "/" + day + "/" + year;

**this**.longDate = monthName + " " + day + ", " + year;

}

**public** String getShortDate() {

**return** shortDate;

}

**public** String getLongDate() {

**return** longDate;

}

**private** String getMonthName(String month) {

String monthName = "";

**switch**(month) {

**case** "01":

monthName = "January";

**break**;

**case** "02":

monthName = "February";

**break**;

**case** "03":

monthName = "March";

**break**;

...

**case** "12":

monthName = "December";

**break**;

}

**return** monthName;

}

}

# Accessing Objects via Reference Variables

|  |  |
| --- | --- |
| Primitive Types | |
| Code | Memory |
| **int x = 3;** | x=3 |
| **int y = x;** | x=3 and y=3 |
| **y = 5;** | x=3 and y=5 |

The code for the example in this section is found in the *example\_box* package.

Objects and primitive data types are handled differently in memory and it is important to understand the difference. All variables have a *type* (data type). For example:

**double** x = 498.57334;

The type of the variable, *x*, is double, which is a primitive type. The primitive types are: byte, char, short, int, long, float, double, boolean. With primitive types, when we use the assignment operator (“=”), we make a physical copy of the value as shown in the figure on the right.

When the equality operator, “==” is applied to primitive type variables, it returns *true* if the two variables have the same value.

**int** x = 3;

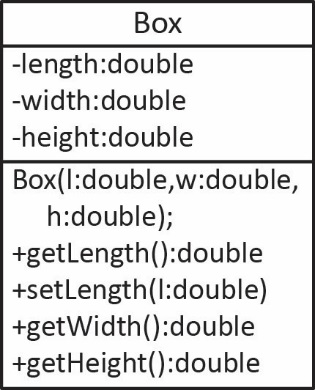
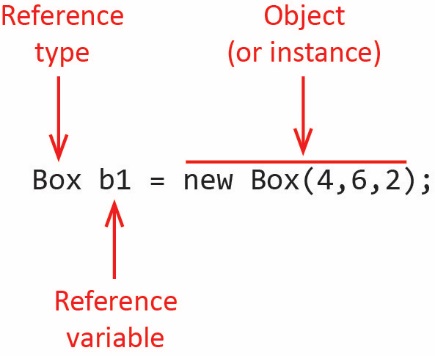
**int** y = 4;

**if**(x==y) {

System.***out***.println(x);

}

Next, let’s contrast how the assignment and equality operators work on objects. Consider the Box class shown in the class diagram on the left, below, and the object creation shown on the right.

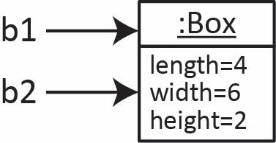
 

The *type* of an *object* is called a *reference type* and is always a class and the corresponding *variable* is called a *reference variable*. As stated earlier, the *reference variable*, *b1,* is not the actual object (although we frequently refer to it that way) but actually contains the memory address of where the actual object is located in memory. In other words, *b1* is a reference (or pointer) to the actual object. The reference variable and the object are stored in different parts of memory.

To reinforce this point, consider the example shown below where we assign one reference variable to another, *b2=b1*. This does not create a copy of the object, it creates another reference to the same object.

|  |  |  |
| --- | --- | --- |
| **Code** | **Memory** | **Description** |
| Box b1 = **new** Box(4,6,2); | Object diagram, reference variable b1 points to a Box in memory. | A new *Box* is created and assigned to the reference, *b1.* |
| Box b2 = b1; | Object diagram, reference variables b1 and b2 both point to the same Box in memory. | A new reference, *b2* is assigned to *b1.* Thus, now there are two references to the same box. In other words, the assignment did not create a copy of the original box. |
| b2.setLength(6); | Object diagram, reference variables b1 and b2 both point to the same Box in memory, after setLength has been called on b2. | Use the reference, *b2* to change the *length* of the box. |
| System.***out***.println(b1.getLength()); // 6  System.***out***.println(b2.getLength()); // 6 | | Use the reference, *b1* to a accessd the *length* of the box. |

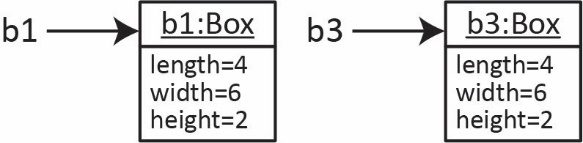
The equality operator, “==” works the same way with reference variables (objects) as it does with primitives, but with one subtle difference. It returns *true* if the two references have the same memory addresses, in other words, if they point to the same object in memory. For example, consider these two statements:

Box b1 = **new** Box(4,6,2);

Box b2 = b1;

Then:

System.***out***.println(b1==b2); // true

This, of course also means that if we have two physically different objects with exactly the same state, then they are not equal. Consider a new *Box, b3* that has exactly the same state as *b1,* then *b3* points to a different place in memory and *b1* and *b3* are not equal:

Box b3 = **new** Box(4,6,2);

System.***out***.println(b1==b3); // false

# The null Keyword

A reference variable can be *null.* What does this mean? Let’s start with a non-software example by considering the image on the right[[3]](#footnote-3).

Many times, we declare and create an object in one line of code. For example:

Account account = **new** Account();

Of course, we can simply declare a reference to object without creating it:

Account account;

In this case, the reference variable, *account* has the special value, *null.* What this means is that it can hold a reference to an *Account* object, but currently it doesn’t. In Java, *null* is a keyword and can be checked for:

**if**(account==**null**) {

System.***out***.println("Account object has not been created");

}

At a later time, you might create an object by invoking the constructor:

account = **new** Account();

Or, you might call a method to build and return an object for you:

account = *buildAccount*();

You can assign a reference variable to be *null* as shown below. This is effectively deleting the object. (Technically, the object is still in memory, but there is no reference variable pointing to it. Eventually, the [*garbage collector*](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))will reclaim the memory). There are situations where this is useful; however, I don’t think we will need to do this in this class.

account = **null**;

If you try to call a method on a reference variable that is *null*, your program will terminate when a *NullPointerException* is thrown. For example, this code:

Account account = **new** Account();

account = **null**;

account.deposit(100.0);

When run, will throw a *NullPointerException*, and the output will look like this:

Account object has not been created

Exception in thread "main" java.lang.NullPointerException

at example\_account.Account1.main(Account1.java:31)

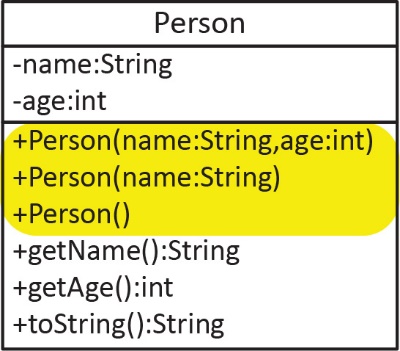
Hopefully, this makes sense: you are trying to call a method (*deposit*), but there is no object, so the program must end. This example is simple, and obvious; however, you will have this happen sometimes in code you write, in less obvious ways than this example. When you get this error, you need to trace backwards in your code to see where it should have been created and add that code.

# Using *this* With Multiple Constructors

A class is not required to have an explicit constructor. If it doesn’t, the compiler inserts a *no-arg constructor*, with no code, into the class. Thus, these two classes are identical:

|  |  |
| --- | --- |
| ***Dog* Class – Implicit Constructor** | ***Dog* Class – Explicit Constructor** |
| **public** **class** Dog {  } | **public** **class** Dog {  **public** Dog() {}  } |

Best practice is to always have at least one explicit constructor defined in a class. Frequently, we provide several constructors to provide flexibility to others who are using our classes.

For example, suppose we want to write a *Person* class that has a *name* and *age* and that a person can be created (constructors) in three ways, by supplying:

* A *name* and an *age*:
* A *name* only, in which case the *age* is set to 0:
* No arguments, in which case the *name* is set to ”Unknown”, and the *age* is set to 0.

Person p = **new** Person("Archie", 33);

Person p = **new** Person("Archie");

Person p = **new** Person();

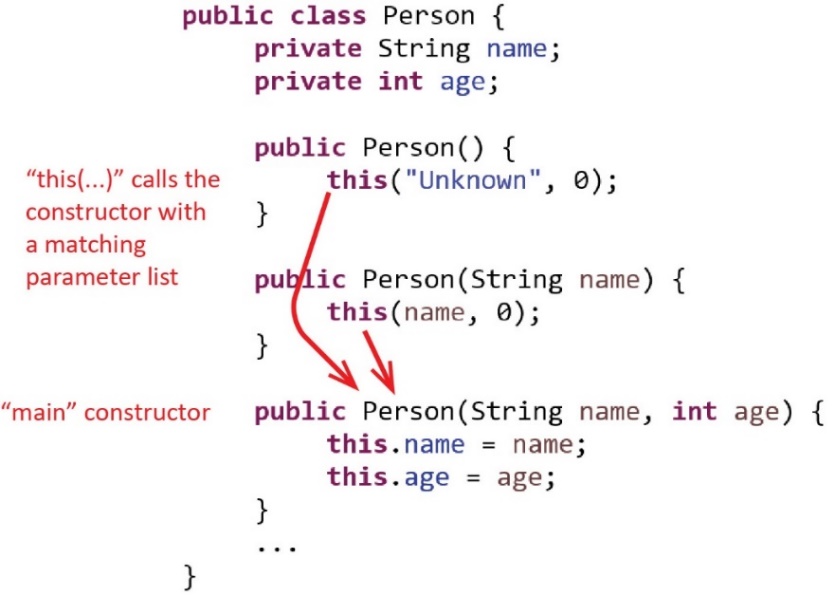
The logical way to code these is shown below. This code is correct; however, the implementation is not a best practice.

|  |  |  |
| --- | --- | --- |
| Person() | Person(String name) | Person(String name, **int** age) |
| **public** Person() {  **this**.name = "Unknown";  **this**.age = 0;  } | **public** Person(String name) {  **this**.name = name;  **this**.age = 0;  } | **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  } |

Java provides a way for a constructor to call another constructor in the same class, with this syntax:

**this**(arg1, arg2, ...);

which must be the first line in the constructor and will call a constructor with the same signature. When you have multiple constructors, best practice is to try to write one “main” constructor, typically the one that accepts the most parameters (*e.g.* Person(name,age) above), and have the other constructors call this one. Best practice for this example is shown in the code below.



Notes:

1. The statement: this(argList) calls the constructor with the matching argument list. Thus, you can’t have two constructors with the same signature.
2. *this* must be the first statement in the constructor, though it can be followed by additional code.
3. We call this *constructor chaining.*
4. It is not always possible, efficient, or understandable to use this technique, but frequently it is.

The solution for the preceding example is found in the *example\_person\_multiple\_constructors* package

## Exercises

1. (Solution in *exercise\_savings\_account* package, *SavingsAccount2* class). Consider the *SavingsAccount* class from an earlier Practice Problem. Add the following constructors, using best practices: (a) accepts only a balance, in which case the name is, “Anonymous”; (b) accepts only an owner, in which case the balance is 0.0; (c) a no-arg constructor that sets the balance to 0.0 an owner to, “Anonymous”. Hint: all three of these constructors should use *this*.
2. (Solution in *exercise\_box* package).Write a *Box* class that stores the length, width, and height of a box. A *Box* can be created three ways: (1) by supplying the length, width, and height; (s) a no-arg constructor, in which case all length, width, and height are all set to 1.0; (3) and if there is only one argument, the length, width, and height are all set to this value. Finally, a *Box* has a *volume* method that returns the volume of the box.

# Passing Objects to Methods

We can pass an object (reference variable) to a method just as we would a primitive; however, it is useful to understand what is occurring in memory. Consider the *Box* class below which has a *mergeBox* method that accepts an instance of a *Box*. As you can see, this method increases the length, width, and height of *this* box by the corresponding values of the box, *b,* that is passed as an argument. The solution is in the *example\_box* package, in the *Box* class.

**public** **class** Box {

**private** **double** length, width, height;

**public** Box(**double** l, **double** w, **double** h) {

length = l; width = w; height = h;

}

**public** **void** mergeBox(Box b) {

length += b.length; width += b.width; height += b.height;

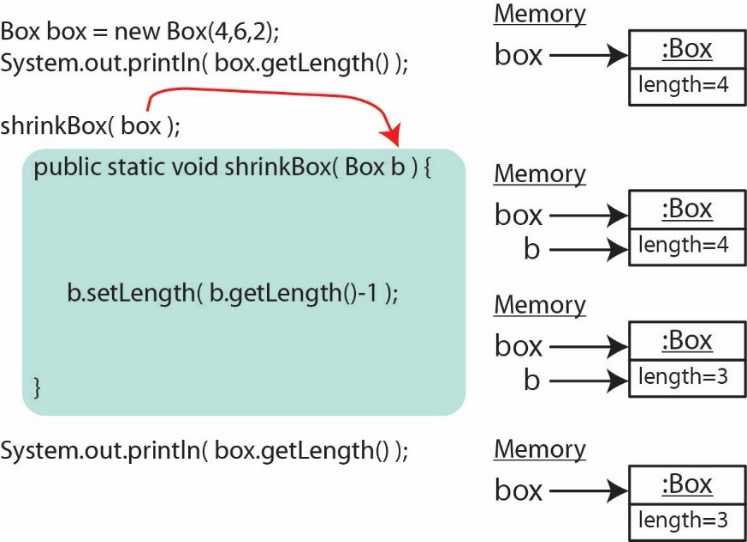
}

...

}

And, we could use this method:

|  |  |  |
| --- | --- | --- |
| Test Code |  | Output |
| Box b1 = **new** Box(2.0,3.0,4.0);  Box b2 = **new** Box(1.0,1.0,2.0);  b1.mergeBox(b2);  System.***out***.println(b1); |  | length=3.0, width=4.0, height=6.0 |

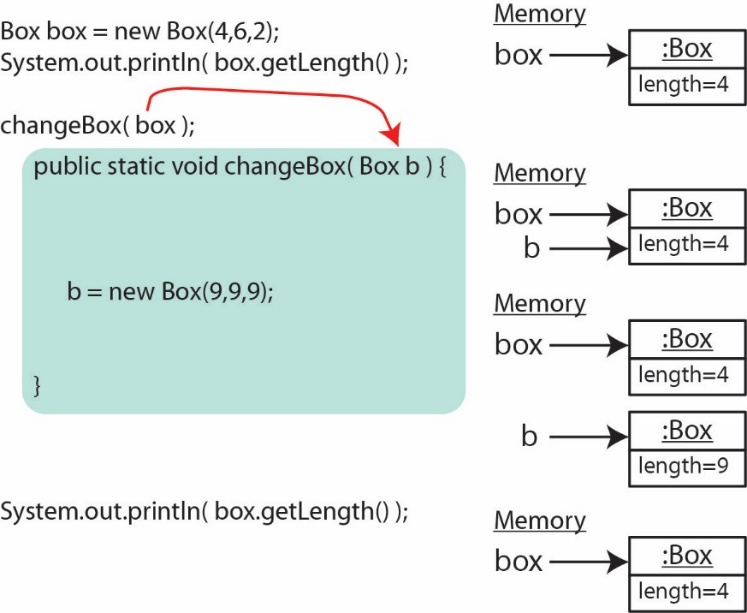
Although the example above was straight forward, it is important to note that all variables in Java are *passed by value*. This means that when you pass an argument to a method, the method receives a *copy* of the argument. In the case of a reference variable, a copy of the pointer to the object is passed. Consider the method below that accepts a *Box* and changes the length of the box.

**public** **static** **void** shrinkBox(Box b) {

b.setLength(b.getLength()-1);

}

On the right, we show what occurs in memory when this method is executed. The calling method defines a *box* reference and when it is passed to the method, a local copy of the reference is made, *b* for use inside the method. As we saw earlier, both *box* and *b* reference the same *Box* object. Inside the method, we use the local reference to manipulate the box. When the method finishes and control returns to the calling program, the reference *b* is no longer available (it is garbage collected), and we continue to use the *box* reference.

Another example is shown below. This method creates a new box, but as we see below, it is short lived. When we return to the calling program, the original reference, *box* still refers to the original box. The new box created in the method is garbage collected when the method ends.

**public** **static** **void** changeBox(Box b) {

b = **new** Box(9,9,9);

}

This example was simply to illustrate how reference variables work. If there were a situation where we want to create a new instance under some condition, we would simply need to return it.

Consider the *changeBox* method below. It accepts a *Box* and if its volume is less than 20, it creates a new box and returns it. Otherwise, the input box is returned, unchanged. We would use this method with code as shown on the right.

|  |  |
| --- | --- |
| *changeBox* Method | Test Code |
| **public** **static** Box changeBox(Box b) {  **if**(b.getVolume()<20.0) {  b = **new** Box(9,9,9);  }  **return** b;  } | Box box = **new** Box(2,3,2);  box = *changeBox*(box); |

A good [reference](https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html), scroll down to “Passing Primitive Data Type Arguments” and “Passing Reference Data Type Arguments”. Another reference, [Pass-By-Value as a Parameter](https://www.baeldung.com/java-pass-by-value-or-pass-by-reference).

## Exercises

1. (Solution in *exercise\_piggy\_bank* package, *PiggyBankVer9* class) Modify the *PiggyBank* class from an earlier Exercise by adding a method, *mergeBank* that accepts another *PiggyBank* object. This method add the quarters, dimes, and nickels from the input bank to this bank. Write some formal test code to test this method.
2. (Solution in *exercise\_savings\_account* package, *SavingsAccount3* class) Modify the *SavingsAccount* class from a previous Exercise by adding a method, *mergeAccount* that accepts another account. If the names (owners) are the same, then the balance of the input account is added to this account’s balance. Hint: make sure you detect if a service fee should be charged, and if *isOverdrawn* should be changed. This takes a bit of thought. Remember, the balance of the input account could be negative. But, a negative balance coming in, doesn’t necessarily make this account overdrawn. There is an easy way to do this by using the deposit and withdraw methods.
3. (Solution in *exercise\_basketball\_player* package) Consider the *BasketBallPlayer* class shown below. Add a method, *hasMorePoints* that accepts another *BasketBallPlayer* instance. This method should return *true* if *this* player has more points than the player that is passed in and *false* otherwise. Informal test code is commented out in *main* below.

**public** **class** BasketballPlayer {

**private** **int** points;

**public** BasketballPlayer(**int** points) {

**this**.points = points;

}

**public** **int** getTotalPoints() {

**return** points;

}

**public** **static** **void** main(String[] args) {

BasketballPlayer p1 = **new** BasketballPlayer(10);

BasketballPlayer p2 = **new** BasketballPlayer(12);

BasketballPlayer p3 = **new** BasketballPlayer(20);

// System.out.println(p1.hasMorePoints(p2));

// System.out.println(p3.hasMorePoints(p2));

}

}

# Arrays of Objects

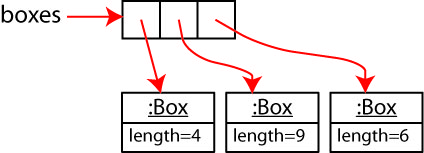
We can create an array of objects just as we would an array of any other type. Consider this “array of Boxes”:

Box[] boxes = new Box[3];

boxes[0] = new Box(4,4,4);

boxes[1] = new Box(9,9,9);

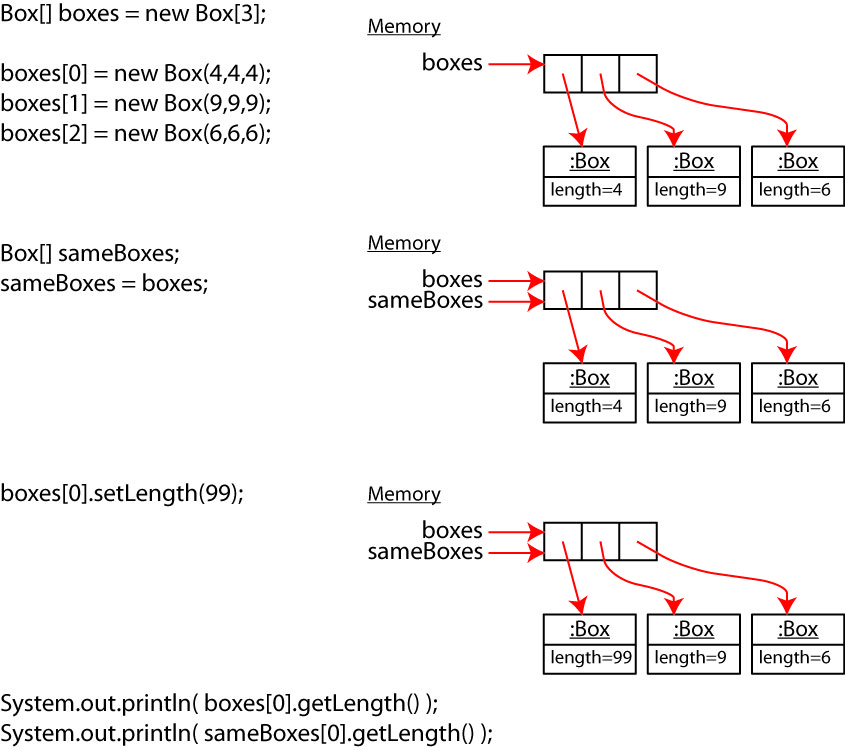
boxes[2] = new Box(6,6,6);

Technically, this “array of Boxes” is really an array of references to *Box* objects. As shown on the right, *boxes* is a reference to an array that resides in memory. The array itself contains references to Box objects that reside elsewhere in memory.

Best Practice – Array variable names should be plural. For example:

Account[] accounts; Employee[] employees; Product[] products; **double**[] dailySales;

**int**[] testScores; String[] names;

Next, as shown in the figure on the right, we consider the case where we have two references to the same array and we use one reference to change an element in the array. As we saw earlier (in a different context), both references reflect the change as (of course) they are both pointing to the same array. Thus, the last two lines produce the same output.

To conclude this section, we consider an example where we write methods that accept and do something with, array(s) of objects. Some will return an array of objects. There is an Exercise that is similar to this example. Typically, I work this on the board, so the solution is found in [Appendix 2](#Appendix_2) (and also in the *example\_box\_utilities* package). The solution uses a *for-each* loop. An explanation of the for-each loop, and when to use it is found in [Appendix 3](#Appendix_3). This example serves three purposes: (a) a review of manipulating arrays, (b) arrays of objects, (c) passing arrays to methods, and returning arrays from methods. Consider the *Box* class below.

**public** **class** Box {

**private** **double** length, width, height;

**public** Box(**double** l, **double** w, **double** h) { length = l; width = w; height = h; }

**public** **double** getLength() { **return** length; }

**public** **double** getWidth() { **return** width; }

**public** **double** getHeight() { **return** height; }

**public** **double** getVolume() { **return** length\*width\*height; }

**public** **void** stretchLenth(**double** percent) { length = length\*(1+percent); }

/\*

\* Determines if b will fit inside this Box, e.g., if all dimensions of b

\* are less than those for this Box.

\*/

**public** **boolean** doesFit(Box b) {

**if**((b.length<**this**.length) &&

(b.width<**this**.width) &&

(b.height<**this**.height)) {

**return** **true**;

}

**else** {

**return** **false**;

}

}

@Override

**public** String toString() {

String msg = String.*format*("len=%.1f, wid=%.1f, ht=%.1f, vol=%.1f",

length, width, height, getVolume());

**return** msg;

}

}

Note: A class as described below, *BoxUtilities* would have static methods instead of instance methods. However, we haven’t talked about static methods (other than for testing) and I don’t want to do that yet.

1. Draw a class diagram for a *BoxUtilities* class with the following members:

* There are no instance variables.
* A no-arg constructor that does nothing, i.e. public BoxUtilities() {}
* A *getAverageVolume* method that accepts an array of *Boxes* and returns the average volume of all the boxes in the array. You can assume the array is full (*i.e.* all elements contain a *Box,* none are null).
* A *stretchAll* method that accepts an array of *Boxes* and a percent (specified as a decimal) and stretches the length of all the boxes in the array by the specified percent.
* A *getBiggest* method that accepts two arrays of *Boxes*. You can assume the two arrays are the same size. This method will return a new array with the biggest boxes, compared by volume, at the corresponding indices. For example:

Notation: (x,y,z) – a box with length=x, width=y, height=z, and volume=x\*y\*z

boxes1 = [ (1,1,1), (7,7,7), (9,9,9) ]

boxes2 = [ (2,2,2), (5,5,5), (6,6,6) ]

biggest = getBiggest(boxes1, boxes2) = [ (2,2,2), (7,7,7), (9,9,9) ]

* A *whichOnesFit* method that accepts two arrays of *Boxes*. You can assume the two arrays are the same size. This method will return a Boolean array that specifies whether the box in the second array fits in the corresponding box in the first array. For example:

boxes1 = [ (1,1,1), (7,7,7), (9,9,9) ]

boxes2 = [ (2,2,2), (5,5,5), (6,6,10) ]

biggest = whichOnesFit(boxes1, boxes2) = [ false, true, false ]

Hint: the *Box* class has a *doesFit(Box b)* method that returns *true* if the input box, *b* fits in this box.

1. Write the *BoxUtilities* class as described above.
2. Write a *BoxUtilitiesTest* class with static methods to test each method.

## Exercises

The solution for this problem is found in the *exercise\_savings\_account\_utilities* package.

1. Consider the *SavingsAccount* class from an earlier Exercise. You will write an *SavingsAccountUtilities* class with the following members:

* There are no instance variables.
* A no-arg constructor that does nothing.
* *getTotalBalance* – accepts an array of *SavingsAccount* objects and returns the sum of the balances over all accounts in the array.
* *applyInterest* – accepts an array of *SavingsAccount* objects and an interest rate and applies this interest rate to all the objects.
* *getAccountsReport* – accepts an array of *SavingsAccount* objects and returns a formatted string showing the name and balance of each account, one per line. The format should be like this:

Accounts Summary

----------------

Num accounts = 3

Total balance = $8,000.00

Accounts:

Owner=Tenesha, bal=$2,000.00

Owner=Niles, bal=$5,000.00

Owner=Xavier, bal=$1,000.00

* *getAccountOwnersList* – accepts an array of *SavingsAccount* objects and returns an array containing just the owner of each account.
* *getSmallestAccounts* – accepts two arrays of *SavingsAccouns*. You can assume the two arrays are the same size. This method will return an array of Account objects with smallest account, compared by balance, at the corresponding indices. For example (this just shows name and balance, you should have the Account object):

account1 = [ (“Bo”,100.0), (“Ann”,500.0), (“Jen”,75.0) ]

account2 = [ (“Sam”,500.0), (“Pete”,200.0), (“Wren”,800.0) ]

smallest = getSmallestAccounts(account1, account2) = [ (“Bo”,100.0), (“Pete”,200), (“Wren”,75.0) ]

1. Draw a class diagram for the *SavingsAccountUtilities* class.
2. Write the *SavingsAccountUtilities* class.
3. Write a *SavingsAccountUtilitiesTest* class with static methods to test each method.

# Immutability

The examples in this section are found in the *example\_immutability* package.

Sometimes it is useful to define a class that doesn’t allow the state of an object to be changed. In other words, the state is set when the object is created, but can never be changed after that. Such a class (object) is called *immutable.* It is probably hard to appreciate why this is useful, but it is a part of defensive design/coding. We can trust the values such an object contains, no one can change it. Obviously, not all classes can be immutable, but there are situations where it is useful.

As an example, the class below is immutable because the instance variables are private and there are no methods (setters or regular methods) that change any of the instance variables.

**public** **class** ImmutableBox {

**private** **double** length, width, height;

**public** **double** getLength() { **return** length; }

**public** **double** getWidth() { **return** width; }

**public** **double** getHeight() { **return** height; }

**public** ImmutableBox( **double** l, **double** w, **double** h ) {

length = l; width = w; height = h;

}

**public** **double** volume() {

**return** length\*width\*height;

}

@Override

**public** String toString() {...}

}

}

This *Student* class below is not immutable because the *getScores* method returns the array of scores, which is a reference variable. Thus, *scores* can be changed.

|  |  |
| --- | --- |
| Class that is not Immutable | Illustration of Mutability |
| **public** **class** Student {  **private** String name;  **private** **int**[] scores;  **public** Student(String name, **int**[] scores) {  **this**.name = name;  **this**.scores = scores;  }  **public** String getName() {  **return** name;  }  **public** **int**[] getScores() {  **return** scores;  }  } | **int**[] scores = {93, 88, 85};  Student s = **new** Student("Larry", scores);  **int**[] scores2 = s.getScores();  scores2[0] = 100; |

Thus, the requirements for a class to be immutable are:

* All instance variables must be private.
* There are no setters (or other methods) that change the values of instance variables.
* No methods can return a reference to an instance variable that is a mutable reference type.

The *String* class is immutable. This means that a *String* can never be changed. When you “change” a string a new *String* is created and assigned to the reference as shown in the figure below. There, we see that the original string no longer has any references to it, so it will be garbage collected.

|  |  |
| --- | --- |
| **Code** | **Memory** |
| String s = "dog"; | Show the memory representation of a string variable, s, pointing to a Dog object.. |
| s += " and cat"; | This, and the next 3 figures illustrate that a method that changes a string without returning the change, loses the change because the changed string dies at the end of the method. Thus, when returning to the calling method, the original string has not been changed. |

|  |
| --- |
| **Method** |
| **public** **static** **void** doesntChangeString(String str){  str += " and cat";  } |

This means that we need to be aware that a *String,* unlike many (mutable) objects cannot be changed in a method. Consider the method shown on the right. Then, consider what happens in memory as the code is executed:

|  |  |
| --- | --- |
| Code | Memory |
| String s = "dog"; | E:\Data-Classes\CS 1302 - Programming 2\notes\02_ch9_Object&Classes\d1.jpg |
| doesntChangeString(s); | This, and the previous figure, and the next 2 figures illustrate that a method that changes a string without returning the change, loses the change because the changed string dies at the end of the method. Thus, when returning to the calling method, the original string has not been changed. |
| // Inside doesntChangeString()  str += " and cat"; | This, and the previous 2 figures, and the next figure illustrate that a method that changes a string without returning the change, loses the change because the changed string dies at the end of the method. Thus, when returning to the calling method, the original string has not been changed. |
| // Back in main | This, and the previous 3 figures, illustrate that a method that changes a string without returning the change, loses the change because the changed string dies at the end of the method. Thus, when returning to the calling method, the original string has not been changed. |

To fix this error, we would simply return the modified string:

**public** **static** String changeString( String s ) {

s += " and cat";

**return** s;

}

# Data Encapsulation

In this section, we discuss *encapsulation.* We have discussed these ideas a little as this chapter progresses, but it is a good time to summarize and say a bit more.

First, the expression, *to* [*encapsulate*](https://www.google.com/search?q=encapsulate)means:

* express the essential features of something succinctly.
* enclose something in or as if in a capsule

When we define a class, we are encapsulating all the relevant features of an object into a class, a single unit. In a more specific way, we refer to *data encapsulation* as when we restrict access to the state of a class. Instead, we only provide methods which can change the values of the state. As stated earlier, *data encapsulation* is a best practice. To make the data in a class encapsulated, we make all instance variables *private* and provide *getters* and *setters* as shown in the table below (and illustrated earlier).

|  |  |
| --- | --- |
| **Not Encapsulated** | **Encapsulated** |
| public class Box {  public double length;  ...  }    Box b = new Box(2,4,6);  b.length = 10; | public class Box {  private double length;  public void setLength(double length) {  if( length > 0 )  this.length = length;  }  public double getLength() {  return length;  }  ...  }  Box b = new Box(2,4,6);  b.setLength(10); |

You may say, “what’s the big deal? Why do I have to do all this extra work writing getters and setters?” The answer is that real systems are composed of many classes written by different teams and developers. Your classes may be used by other developers and *vice versa*. If your *Box* class is not encapsulated, *you* may know not to assign a negative value to the length, but others using your class may not know that. It is better to practice *defensive programming*, to take steps to make sure others use your classes correctly. One way to do this is to *protect* (encapsulate) the *state* of your classes. In addition to inadvertent misuse of a class, encapsulation also protects against deliberate misuse of a class. For example, you wouldn’t define a *BankAccount* class with a public *balance* instance variable because then clients could change the balance to any value they wanted. And, suppose you needed to log every transaction (change in balance) in a database. If you had a public *balance* instance variable, and a method to update the database, you would be requiring someone using your class to remember that both steps are necessary. *Business rules*, such as this,should be encapsulated by writing methods that carry them out. So, instead, you define the balance as private, and define a *deposit* method which updates the balance, and logs the amount of the deposit in a database and updates the private *balance*.

I use the following analogy to illustrate why data encapsulation is important. Consider the not-encapsulated *You* class:

**public class You {**

**public double wallet;**

**}**

By defining your wallet as public, when you checkout at the grocery, the cashier just reaches in your pocket and pulls your wallet out, and takes the required amount of money.

But that, of course is not the way things work. Your wallet is private! Now, the cashier says, “$10 please” (they are calling your *payCashier* method), and you reach into your wallet and pull out and hand them the required amount of money.

**public class You {**

**private double wallet;**

**public double payCashier(double amount) {**

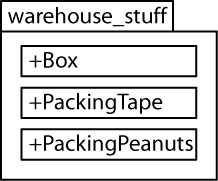
**// take money out of wallet and hand to cashier**

**}**

**}**

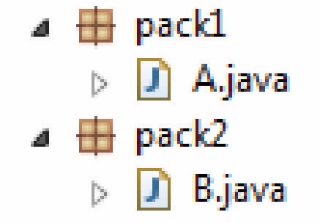
As a final note here, we introduce a term, *read-only* property, which we have discussed before, without using that name. When we want to allow clients to retrieve the value of a property, but not change it, then we supply a getter, but no setter. This is called a *read-only* property. When designing systems, we should defer to making properties read-only, unless there is an important reason for them to be changed through a setter. For example, a *Student* class would define a *studentNumber* property which would probably be read-only.

# Packages & Accessibility (Visibility) of Classes & Members

A *package* is a way to group related classes. The UML *package diagram* shown on the right shows a package named *warehouse\_stuff* which contains three classes: *Box, PackingTape,* and *PackingPeanuts.* A package not only provides a way to organize classes, it also provides *protection* as we will see below.

We declare a package with a *package statement* as the first line in a file. By convention, package names are all lower-case with an underscore separating words. For example:

|  |  |  |
| --- | --- | --- |
| **Box.java** | **PackingTape.java** | **PackingPeanuts.java** |
| package warehouse\_stuff;  public class Box {  ...  } | package warehouse\_stuff;  public class PackingTape {  ...  } | package warehouse\_stuff;  public class PackingPeanuts {  ...  } |

All classes in a package must be inside a folder with the same name as that of the package. In the figure on the right, *A.java* is in the *pack1* package, thus, it is in a *pack1* folder on disk.

If you have a class, *A,* that wants to use a class, *B,* which is in another package, *pack2,* then it must be imported as shown in the code below.

|  |  |
| --- | --- |
| A.java | B.java |
| **package** pack1;  **import** pack2.B;  **public** **class** A {  **public** **static** **void** main( ... ) {  B b = **new** B();  ...  }  } | **package** pack2;  **public** **class** B {  ...  } |

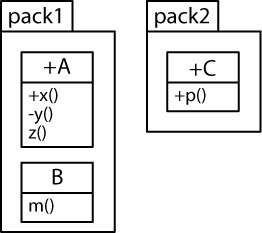
You can import all classes in a package with this statement: import pack.\*; however, best practice is to only import the specific class(es) you need.

There are 2 visibility (accessibility) modifiers for classes as shown in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| **UML** | **Java** | **Example** | **Meaning** |
| + | public | **public class Box { ... }** | The class is available in any package |
| [blank] | [blank] | **class Box { ... }** | The class is available only in containing package |

The “[blank]” visibility is called *default* visibility or *package-level* visibility. There are 4 visibility modifiers for class members (methods and instance variables). In order of most to least visible, they are:

|  |  |  |
| --- | --- | --- |
| **UML** | **Java** | **Meaning** |
| + | public | Available in any class |
| # | protected | [We learn about this in Ch 3] |
| [blank] | [blank] | Available in any class in the same package. This is called *default* visibility. |
| - | private | Available only in the class. |

Classes and methods can access classes and methods in other packages if they are imported and if the accessibility allows it. For example, consider the packages and classes shown in the package diagram on the right and note the following:

* Class *A* can access class *C* because *C* is public. It can also access class *B* because it is in the same package.
* Class *B* can access classes *A* and *C*.
* Class *C* can access class *A*, but not *B* because class *B* has package-level accessibility.
* Methods in *C* can access the *x* method in *A*, but not the *y* method because it is private, and not *z* because it has package-level accessibility.
* Methods in *B* can access the *x* and *z* methods in *A* because they are in the same package.
* Methods in *A* can access any methods in *B* or *C*

We can also define subpackages. For example:

package pack1.subpack;

public class A {...}

In this case, class *A* would be defined in a *subpack* folder and the *subpack* folder would be inside the *pack1* folder. For a class in another package to access class *A,* you need to import the package and subpackage. For example:

package another\_package;

import pack1.subpack.A;

...

# Static Variables & Methods

The examples in this section are found in the *example\_student\_static* package.

## Static Variables

A *static variable* is **shared by all instances of a class**. It is not an *instance variable*, it is a *class variable*. An *instance variable* belongs to a specific object; a *static (class*) variable belongs to the class.

Example 1 – Suppose we have a situation where we want every instance of a class to know how many instances of that class have been created. In the example below, we have a *Student* class that has a static variable, *numStudents* which is incremented each time an object is created.

|  |  |
| --- | --- |
| *Student* Class | Test Code |
| **public** **class** Student {  **private** String name;  **private** **static** **int** *numStudents*;    **public** Student(String name) {  **this**.name = name;  *numStudents*++;  }  ...  } | Student s1 = **new** Student("Maria");  System.***out***.println(s1); // name=Maria, numStudents=1  Student s2 = **new** Student("Zoe");  System.***out***.println(s2); // name=Zoe, numStudents=2  Student s3 = **new** Student("Deion");  System.***out***.println(s3); // name=Deion, numStudents=3 |

From a design standpoint, this approach is used because the *Student* objects themselves need to know how many other students there are. This situation rarely, if at all, occurs.

Next, we present an example of using a static variable inappropriately. The problem occurs when an instance variable is needed, but, incorrectly, it is declared static. However, subsequent code treats it as an instance variable.

Example 2 – Consider a *Student* class with *name* and *age* properties; however, we define the *age* as static.

|  |
| --- |
| *Student* Class |
| **public** **class** Student {  **private** String name;  **private** **static** **int** *age*;    **public** Student(String name, **int** initAge) {  **this**.name = name;  *age* = initAge;  }    **public** **void** setAge(**int** newAge) {  *age* = newAge;  }  ...  } |

Now, consider the sample code and the output below. First, we create a student and then display them:

Student s1 = **new** Student("Sly", 33);

System.***out***.println(s1); // name=Sly, age=33

Next, we create another student, with a different age. However, there is only one *age* variable and it is shared by all instances of the class. So, when we print the two students, they both have the same age.

Student s2 = **new** Student("Wren", 44);

System.***out***.println(s1); // name=Sly, age=44

System.***out***.println(s2); // name=Wren, age=44

And of course the same thing happens when we use *setAge*.

s2.setAge(77);

System.***out***.println(s1); // name=Sly, age=77

System.***out***.println(s2); // name=Wren, age=77

There is a use for static variables; however, you will never need a static variable in this class. Do not use them in this class.

## Constants – *final*

A variable whose value cannot change is called a *constant.* We use the *final* variable modifier to indicate a variable (instance variable or local variable) is a constant. A *final* variable can only be assigned a value one time. A *final* instance variable must be initialized in a constructor or in its declaration. In the example below on the left, a *Student* has a *ssn* which is *final* and initialized in the constructor. Alternately, as shown on the right below, *ssn* can be initialized in its declaration. Any other line of code that attempts to assign it a value will not compile.

|  |  |
| --- | --- |
| **Initialize in Constructor** | **Initialize in Declaration** |
| **public** **class** Student3 {  **private** **final** String ssn;  **private** String name;  **private** **int** age;    **public** Student3(String ssn,  String name, **int** age) {  **this**.ssn = ssn;  **this**.name = name;  **this**.age = age;  }  ... | **public** **class** Student3a {  **private** **final** String ssn = "222-31-9536";  **private** String name;  **private** **int** age;    **public** Student3a(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  ... |

A good question is why would you make an instance variable *final*? The effect of *final* is essentially the same as defining an *immutable* property: an instance variable declared without *final* and not supplying any methods that could change the value. For example, in the example above, if there is no method in the *Student* class that changes *ssn* then we can remove the *final* in its declaration. There are several reasons that *final* can be useful:

* If you want an immutable property in a class, marking it *final* makes this explicit, and it will be in the first section of the class so it is quicker to learn that information. It increases readability.
* *final* variables are treated differently in the JVM and this leads to safer operations in threaded applications.

However, if a final field points to an object, the object itself can be changed, if it is mutable. For example, the *Student* class below has an *age* that can be changed with *setAge*. In *main,* a *Student* object is created and accessed by a *final* reference variable, *s1*. As shown, although we can’t change what object *s1* points to, we can change the object itself.

|  |  |
| --- | --- |
| **public** **class** Student4 {  **private** String name;  **private** **int** age;    **public** Student4(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }    **public** **void** setAge(**int** age) {  **this**.age = age;  }  ... | **public** **static** **void** main(String[] args) {  **final** Student4 s1 = **new** Student4("Maria", 33);  s1.setAge(77);  } |

## Class Constants – *static final*

A *class constant* is a variable whose value cannot be changed and is shared by all instances of a class. A class constantshould have any visibility other than *private.* For example, the [*Math*](https://docs.oracle.com/javase/9/docs/api/java/lang/Math.html)class declares these class constants:

**public** **static** **final** **double** ***E*** = 2.7182818284590452354;

**public** **static** **final** **double** ***PI*** = 3.14159265358979323846;

To access a static member of a class, we use: *ClassName.nameOfMember*. For example

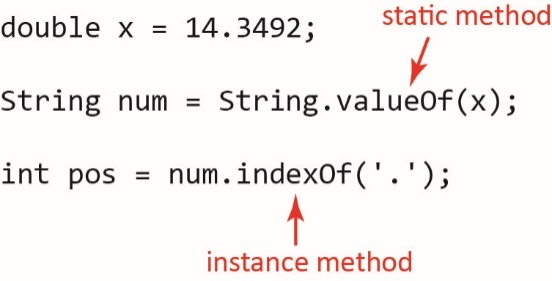
Math.E or Math.PI

The naming convention for constants is to use all capital letters, with words separated by an underscore. For example:

**public** **static** **final** **int** ***MAX\_INTERVAL*** = 30;

## Static Methods

A *static method* is shared by all instances of a class; thus, it cannot depend on the values of instance variables for a particular object. For example, the *String* class defines the static method, *valueOf* that converts an integer (or double, boolean, *etc*) into a string. We use the class name to reference a static method. For example:



The *Math* class is a *utility* class. All the methods it defines are static and you cannot create an instance of the class. You can simply use its (static) methods to do various mathematical computations. For example:

**double** x = Math.*abs*(-4.23);

**double** y = Math.*pow*(4,2);

**double** z = Math.*sqrt*(45.76);

In this class, you will NEVER define static methods except in test classes. For example:

**public** **class** BoxTest {

**public** **static** **void** main(String[] args) {

*boxTest1*();

}

**public** **static** **void** boxTest1() {

Box b = **new** Box(2,3,4);

System.***out***.println(b.volume());

...

}

}

## References

* <https://www.baeldung.com/java-static>
* <https://www.geeksforgeeks.org/static-keyword-java/>
* <https://stackoverflow.com/questions/21155438/when-to-use-static-variables-methods-and-when-to-use-instance-variables-methods>
* <https://dzone.com/articles/final-keyword-and-jvm-memory-impact>
* <https://ocramius.github.io/blog/when-to-declare-classes-final/>

# Chapter Summary

This is a review of the concepts in Chapter 1. It should include some short code snippets for most topics, but for now, it doesn’t.

**Sections 1-3**

*Classes* model things (*objects*) – more on this in a minute. A class is a blueprint for making objects. We use classes to create objects. We program with objects.

Classes have *state* (*instance variables, attributes, properties, fields*) and *behavior* (*methods, services, responsibilities*). *State* is the data that defines what it means to be a particular type of object. *Behavior* is the methods that specify what you can do with an object.

**Sections 4-5**

A class should have a *constructor* which is used by client code to create objects. Constructors generally have parameters, one for each instance variable (but of course, there are many exceptions to this).

The *members* of a class are: instance variables, methods, constructors.

**Sections 6-7**

All instance variables should be *private*. *private* means that the variable is only accessible *inside* the class.

Almost all instance variables will have a *getter*. A *getter* simply returns the value of the instance variable. If you want to also provide a way to change the value of an instance variable, you would provide a *setter*, which accepts the new values and assigns it to the instance variable.

An instance variable that is an array is used so that the object has an “array” or properties of the same type. In this case, an *indexed getter* is used to return a particular value in the array. An indexed setter would accept the new values and the location in the array where the new value will be placed.

**Sections 8-9**

Use *this* to differentiate between an instance variable and a parameter or local variable. We typically use it in the constructor where we accept the initial value of the instance variable with a parameter name the same as the instance variable. We use *this.* in front of the instance variable to resolve the difference.

Every class should have a *toString* method that accepts no parameters and returns a string. It should return a string with the values of the instance variables along with labels to make the output meaningful.

**Sections 10-13**

Not on test 1: however, will be summarized here at some other time. Topics: testing, class diagrams, best practices

**Sections 14-15**

Use these String methods: *length, charAt(i), equals, substring(i,j), substring(i), indexOf(ch/str)*.

Use *String.format(“format string”, var1, var2, …)* to return a formatted string. The *format string* is exactly what you want the return string to look like, except in place of the values of variables, you use *format specifiers*. Use: “%s”, “%d”, “%,.xf” for strings, ints, and doubles, respectively, where *x* is the number of decimal places.

*Helper methods* are a best practice. This says we should replace long methods with calls to *helper methods* that do a portion of the work the long method did.

**Sections 16-17**

A variable that points (refers) to an object is called a *reference variable* and its data type (a class) is called a *reference type.* A reference variable stores the memory address of the actual object in memory and is stored in a section of memory called the *stack.* The object it points to is stored in a section of memory called the *heap.*

When you copy a reference variable (via assignment), you create another reference to the same object in memory.

When you declare a reference variable, without creating an object, the reference variable has the special value, *null.* Also, *null* can assigned to a reference variable, explicitly removing an object from be referenced by the variable. We can also check to see if a reference variable is null. Calling a method on a reference variable that is null results in a *null pointer exception.*

**Sections 18**

If a class doesn’t explicitly define a constructor, the compiler automatically adds a *no-arg constructor* that does nothing (behind the scenes it creates the object in memory.

A class can have any number of constructors. Best practice is to try to write one “main” constructor, typically the one that accepts the most parameters, or does the most work, and have the other constructors call this one using the *this* keyword.

To call another constructor with *this*, it must be the first line of code in the constructor, and it is followed by an arguement list. Thus, it looks similar to a method call where the name of the method is *this*. At run-time, when such a constructor is called, the JVM calls the constructor with the matching parameter list, including number of parameter and data type.

**Sections 19-20**

Methods can define parameters that are objects. The examples we saw where a class, *X*, has a method that defines a parameter of the same type, *X.*

An *array of object* is actually a reference variable that points to an array object in memory, where each cell in the array refers to an object in another place in memory.

We can write methods that process an array(s) of objects.

**Sections 21-24**

Not on test 1: however, will be summarized here at some other time. Topics: immutability, encapsulation, packages & visibility, static variables, static methods

**Appendix 1**

This needs a summary of the string methods.

**Appendix 3**

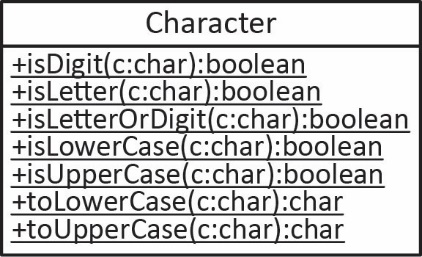
A *for-each* loop is a way to iterate over an array without an index, that visits every element from beginning to end.

Appendix

1. String & Characters

This section is a review of the *String* and *Character* classes’ methods. String manipulation is a fundamental, essential programming skill. String manipulation will be required in most homework assignments and tests. Make sure you can use any of the methods below to solve problems.

* 1. Characters

[*Character*](https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html) is a class in the Java API which defines a number of useful static methods as shown in the class diagram on the right. It should be self-explanatory as to what the methods do. Example:

**char** code = 'a';

**boolean** isChar = Character.*isLetter*(code);

code = Character.*toUpperCase*(code);

Note: to invoke a static method, we use the class name: ClassName.staticMethod(arguments)

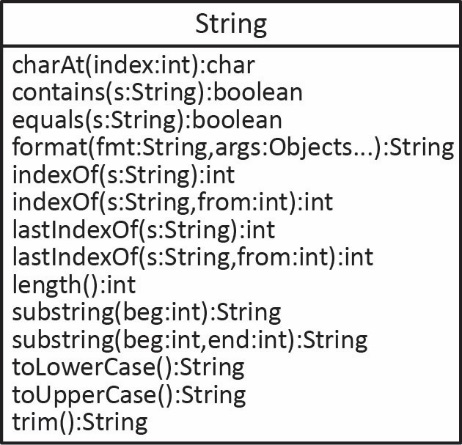
Comparing Characters – Since characters are represented internally as integers, we can compare characters with: =, !=, <, <=, >, >=. Note: upper-case letters occur before lower-case. From smallest to largest[[4]](#footnote-4): (A,B,…,Z,…a,b,…z) For example:

**if**(code=='b') {

count++;

}

* 1. Strings

**[*String*](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) is a class in the Java API. Some of the common methods are shown in the class diagram on the right. We will review some of them below. We usually create a *String* by assigning it to a *string literal*:

String s = "buffalo";

However, we can use the constructor:

String s = **new** String("buffalo");

We use an *escape sequence* to represent special characters. In Java (and many languages) this is a backslash followed by a character (or digits). Several useful escape sequences are:

|  |  |
| --- | --- |
| **Escape Sequence** | **Name** |
| \n | New line |
| \t | Tab |
| \\ | Backslash |
| \” | Double quote |

For example:

|  |  |
| --- | --- |
| Code | Output |
| String msg = "The \"first\"\nThe second ";  System.***out***.println(msg); | The "first"  The second |

The table below shows some of the basic *String* methods. *length* and *charAt* are useful as they allow you to iterate over each character in a string.

|  |  |
| --- | --- |
| **Method** | **Description** |
| length():int | Returns the length of the string, the number of characters. |
| charAt(i:int):char | Returns the character at index *i* in the string. |
| trim():String | Returns a string whose value is this string with any leading or trailing whitespace removed. Useful when reading data a user has supplied. |
| toUpperCase():String | Returns a string whose value is this string and with all the characters being converted to upper case. |
| toLowerCase():String | Returns a string whose value is this string and with all the characters being converted to lower case. |
| concat(s:String):String | Returns a string whose value is this string with *s* concatenated to the end. Usually we just use the “+” operator to concatenate strings. Both of these are equivalent:  String s1 = "anteater";  String s2 = "buffalo";  String s3 = s1 + s2;  String s4 = s1.concat(s2); |

|  |
| --- |
| **public** **class** Word {  **private** String word;    **public** Word(String word) {  **this**.word = word;  }  } |

Example – Suppose we have a *Word* class as shown on the right that stores a *word* string. Write a method, *countUpperCase* for this class that returns the number of upper-case letters in the instance variable, *word*.

Algorithm: loop through the characters in the *word* and ask each one if it is upper case.

Solution:

**public** **int** countUpperCase() {

**int** count = 0;

**for**(**int** i=0; i<word.length(); i++) {

**if**(Character.*isUpperCase*(word.charAt(i))) {

count++;

}

}

**return** count;

}

* 1. String Comparison Methods

These are several comparison methods in the *String* class. We discuss them after this summary.

|  |  |
| --- | --- |
| **Method** | **Description** |
| equals(s:String):boolean | Returns *true* if *s* has exactly the same contents as this string. |
| contains(s:String):boolean | Returns *true* if *s* is in this string. Note: there is not an overload that accepts a character. |
| compareTo(s:String):int | Returns a negative integer if this string is lexicographically less than *s,* a positive integer if this string is greater than *s,* and 0 if they are the same. |

To see if two strings have the same contents, use the *equals* method:

**if**(name1.equals(name2)) {

System.***out***.println("Names the same");

}

Do NOT use “==” to compare the contents of two strings. Since strings are objects (reference types), when you use “==”, the JVM checks to see if the two *String* objects occupy the same location in memory, not if they have the same contents. For example:

s1 = "help";

s1 += "er";

s2 = "helper";

System.***out***.println(s1==s2); // false

System.***out***.println(s1.equals(s2)); // true

The *contains(s:String)* method is used to see if the input string, *s* is contained in the string that is calling this method. For example, consider the *Word* class above. Add a method, *containsDotCom* that returns *true* if the *word* instance variable contains the string, “.com”, and *false* otherwise. The solution is shown below on the left. However, the *contains* method returns a Boolean (which is the answer), so we can write it as shown on the right.

|  |  |  |
| --- | --- | --- |
| OK |  | Preferred |
| **public** **boolean** containsDotCom2() {  **if**(word.contains(".com")) {  **return** **true**;  }  **return** **false**;  } |  | **public** **boolean** containsDotCom() {  **return** word.contains(".com");  } |

Example – Consider the *Word* class above. Add a method, *countVowels* that returns the number of vowels in *word*:

1. A fairly obvious algorithm is to loop over the characters in *word* and check to see if it is: a,e,i,o, or u.

**public** **int** countVowels2() {

**int** count = 0;

**for**(**int** i=0; i<word.length(); i++) {

**char** c = word.charAt(i);

**if**(c=='a' || c=='e' || c=='i' || c=='o' || c=='u') {

count++;

}

}

**return** count;

}

1. Another approach is to define a string that contains the vowels:

String vowels = "aeiou";

Then, loop over the characters and ask each one if it is contained in *vowels*:

**public** **int** countVowels() {

String vowels = "aeiou";

**int** count = 0;

**for**(**int** i=0; i<word.length(); i++) {

String c = word.charAt(i) + "";

**if**(vowels.contains(c)) {

count++;

}

}

**return** count;

}

Note that the *contains* method accepts a *String*; there is no overloaded version that accepts a *char.* Thus, one way to turn the current character into a string is to concatenate an empty string onto the end of it as shown above. Another is:

String c2 = String.*valueOf*(word.charAt(i));

Characters are represented by [Unicode](https://en.wikipedia.org/wiki/List_of_Unicode_characters#Basic_Latin) values (integers). The lexicographic order of these are: 0, 1, 2, …, A, B, C, …, a, b, c. Thus, the upper-case characters occur before the lower-case characters.

1. The *String* class’s *compareTo* method is an important method. When you sort an array of strings, Java’s sort method uses the *compareTo* method. We don’t explicitly use it in code we write very often. However, it will come up later in the semester, so we will look at how it works for the *String* class as an aid to when we cover it more thoroughly later.

1. The *compareTo(s:String)* method starts at the beginning of this string and the input string, *s,* and compares them character by character. If the two strings are the exactly the same, it returns 0. If they are different, then the first pair of characters that are different are used to determine the return value. It returns a negative integer if *this* string occurs before the argument, *s;* and it returns a positive integer if *this* string occurs after *s.* For example.

String s1 = "ant";

String s2 = "fox";

String s3 = "alpaca";

String s4 = "ant";

System.***out***.println(s1.compareTo(s2)); // -5, ‘a’ is 5 characters before ‘f’

System.***out***.println(s2.compareTo(s1)); // 5, ‘f’ is 5 characters after ‘a’

System.***out***.println(s1.compareTo(s3)); // 2, ‘n’ is 2 characters after ‘l’

System.***out***.println(s1.compareTo(s4)); // 0, the strings are the same

1. As noted above, the upper-case characters occur before the lower-case letters. For example:

String s1 = "fox";

String s2 = "Fox";

System.***out***.println(s1.compareTo(s2)); // 26

* 1. Substring Methods

The *String* class has methods for returning a substring:

|  |  |
| --- | --- |
| **Method** | **Description** |
| substring(beg:int):String | Returns the substring in this stringthat beings at index *beg* and extends to the end of the string. |
| substring(beg:int,end:int):String | Returns the substring in this stringthat begins at index *beg* and extends to the character at index *end-1*. Thus, the length of the returned string will be *end-beg*. |

For example:

String s1 = "anteater";

Shows the string "anteater" with an index under each character. For example: a is at index 0, n is at index 1, etc. Highlights the portion of the string between indices 3 through 7, "eater".String s2 = s1.substring(3); // "eater"

Shows the string "anteater" with an index under each character. For example: a is at index 0, n is at index 1, etc. Highlights the portion of the string between indices 3 through 5, "eat".String s3 = s1.substring(3,6); // "eat"

* 1. Conversion of other Types to String

To convert a string which is an integer (double), use one of these below. If the string is not a number (or the wrong type) a run-time error will result.

String s1 = "48";

**int** x = Integer.*parseInt*(s1);

String s2 = "933.92";

**double** y = Double.*parseDouble*(s2);

Additionally, there are two other techniques to convert a number, character, or boolean to a string:

|  |  |  |
| --- | --- | --- |
| Number to String | Character to String | Boolean to String |
| **double** x = 43.42;  String s1 = String.*valueOf*(x);  // Or  String s2 = "" + x; | **char** c = 'B';  String s3 = String.*valueOf*(c);  // Or  String s4 = "" + c; | **boolean** isValid = **true**;  String s5 = String.*valueOf*(isValid);  // Or  String s6 = "" + isValid; |

* 1. String Location Methods

These methods in the *String* class return the location of an input character or string inside this string

|  |  |
| --- | --- |
| **Method** | **Description** |
| indexOf(c:char):int  indexOf(s:String):int | Returns the index of the first occurrence of *c* (*s*)in this stringor -1 if not found. |
| indexOf(c:char,from:int):int  indexOf(s:String,from:int):int | Returns the index of the first occurrence of *c* (*s*)in this stringwhich occurs at or after the index *from,* or -1 if not found. |
| lastIndexOf(c:char):int  lastIndexOf(s:String):int | Returns the index of the last occurrence of *c* (*s*)in this stringor -1 if not found. |
| lastIndexOf(c:char,from:int):int  lastIndexOf(s:String,from:int):int | Returns the index of the last occurrence of *c* (*s*)in this stringsearching backwards starting at *from.* |

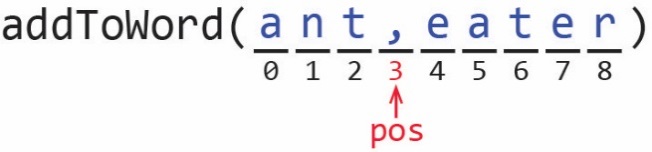
For example:

|  |  |
| --- | --- |
| Example | Example Code |
| String s1 = "anteater";   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | a | n | t | e | a | t | e | r | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | **int** a = s1.indexOf('e'); // 3  **int** b = s1.indexOf("ate"); // 4  **int** c = s1.indexOf("bill"); // -1  **int** d = s1.indexOf("te", 3); // 5  **int** e = s1.lastIndexOf("te"); // 5  **int** f = s1.lastIndexOf('a',3); // 0 |

Example – Consider the *Word* class above. Add a method, *addToWord* in the *Word* class that accepts a string that may contain a comma. If it does, then it should remove the comma and add it to *word*; otherwise, it should simply add the argument to *word*. For example:

|  |  |
| --- | --- |
| **Case 1: Contains Comma** | **Case 2: Doesn’t Contain Comma** |
| Word w1 = **new** Word("Orange");  w1.addToWord("Peel,Now"); // "OrangePeelNow" | Word w1 = **new** Word("Orange");  w1.addToWord("Peel"); // "OrangePeel" |

The solution is shown below:

**public** **void** addToWord(String str) {

**int** pos = str.indexOf(',');

**if**(pos != -1) {

String word1 = str.substring(0, pos);

String word2 = str.substring(pos+1);

word += word1 + word2;

}

**else** {

word += str;

}

}

Alternate Solution:

**public** **void** addToWord2(String str) {

**if**(str.contains(",")) {

**int** pos = str.indexOf(',');

String word1 = str.substring(0, pos);

String word2 = str.substring(pos+1);

word += word1 + word2;

}

**else** {

word += str;

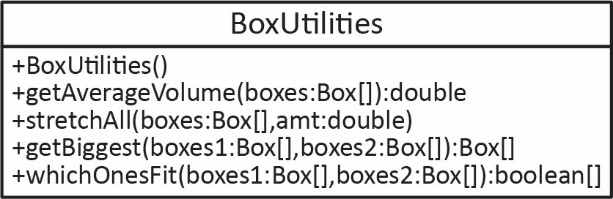
}

}

Example – Consider the *Word* class above. Add a method, *getUniqueCharacters* in the *Word* class that returns a string with exactly one instance of each character in *word*. For example, if *word=*“abbcKczza”, then the method returns “abcKz”.

|  |  |
| --- | --- |
| **Solution** | **Alternate Solution** |
| **public** String getUniqueCharacters() {  String result = "";  **for**(**int** i=0; i<word.length(); i++) {  String c = String.*valueOf*(word.charAt(i));  **if**(!result.contains(c)) {  result += c;  }  }  **return** result;  } | **public** String getUniqueCharacters2() {  String result = "";  **for**(**int** i=0; i<word.length(); i++) {  **char** c = word.charAt(i);  **if**(result.indexOf(c)==-1) {  result += c;  }  }  **return** result;  } |

1. The *BoxUtilities* Class



**public** **class** BoxUtilities {

**public** BoxUtilities() {}

**public** **double** getAverageVolume(Box[] boxes) {

**double** sum = 0.0;

**for**(Box b : boxes) { // see Appendix 4 for explanation of for-each loop

sum += b.getVolume();

}

**return** sum/boxes.length;

}

**public** **void** stretchAll(Box[] boxes, **double** percent) {

**for**(Box b : boxes) {

b.stretchLenth(percent);

}

}

**public** Box[] getBiggest(Box[] boxes1, Box[] boxes2) {

Box[] biggest = **new** Box[boxes1.length];

**for**(**int** i=0; i<boxes1.length; i++) {

Box b1 = boxes1[i];

Box b2 = boxes2[i];

**if**(b1.getVolume()>b2.getVolume()) {

biggest[i] = b1;

}

**else** {

biggest[i] = b2;

}

}

**return** biggest;

}

**public** **boolean**[] whichOnesFit(Box[] boxes1, Box[] boxes2) {

**boolean**[] whichFit = **new** **boolean**[boxes1.length];

**for**(**int** i=0; i<boxes1.length; i++) {

Box b1 = boxes1[i];

Box b2 = boxes2[i];

whichFit[i] = b1.doesFit(b2);

}

**return** whichFit;

}

}

Note, you could write the highlighted line above like this:

**if**(b1.doesFit(b2)) {

whichFit[i] = **true**;

}

**else** {

whichFit[i] = **false**;

}

**public** **class** BoxUtilitiesTest {

**public** **static** **void** main(String[] args) {

*testGetAverageVolume*();

*testStretchAll*();

*testGetBiggest*();

*testWhichOnesFit*();

}

**private** **static** **void** testGetAverageVolume() {

System.***out***.println("Test getAverageVolume(boxes)");

Box[] boxes1 = *createBoxes1*();

BoxUtilities bu = **new** BoxUtilities();

**double** avgVol = bu.getAverageVolume(boxes1);

String msg = String.*format*("Avg vol=%.2f", avgVol);

System.***out***.println(msg);

}

**private** **static** **void** testStretchAll() {

System.***out***.println("\nTest stretchAll(boxes,0.1)");

Box[] boxes1 = *createBoxes1*();

BoxUtilities bu = **new** BoxUtilities();

System.***out***.println("before stretch");

**for**(Box b : boxes1) System.***out***.println(b);

bu.stretchAll(boxes1, 0.1);

System.***out***.println("after stretch");

**for**(Box b : boxes1) System.***out***.println(b);

}

**private** **static** **void** testGetBiggest() {

System.***out***.println("\nTest getBiggest(boxes1,boxes2)");

Box[] boxes1 = *createBoxes1*();

Box[] boxes2 = *createBoxes2*();

BoxUtilities bu = **new** BoxUtilities();

System.***out***.println("boxes1=");

**for**(Box b : boxes1) System.***out***.println(b);

System.***out***.println("boxes2=");

**for**(Box b : boxes2) System.***out***.println(b);

Box[] biggest = bu.getBiggest(boxes1, boxes2);

System.***out***.println("biggest=");

**for**(Box b : biggest) System.***out***.println(b);

}

**private** **static** **void** testWhichOnesFit() {

System.***out***.println("\nTest whichOnesFit(boxes1,boxes2)");

Box[] boxes1 = *createBoxes1*();

Box[] boxes2 = *createBoxes2*();

BoxUtilities bu = **new** BoxUtilities();

System.***out***.println("boxes1=");

**for**(Box b : boxes1) System.***out***.println(b);

System.***out***.println("boxes2=");

**for**(Box b : boxes2) System.***out***.println(b);

**boolean**[] whichFit = bu.whichOnesFit(boxes1, boxes2);

System.***out***.println("whichFit=");

**for**(**boolean** b : whichFit) System.***out***.println(b);

}

**private** **static** Box[] createBoxes1() {

Box[] boxes1 = **new** Box[3];

boxes1[0] = **new** Box(4,4,4);

boxes1[1] = **new** Box(9,9,9);

boxes1[2] = **new** Box(6,6,6);

**return** boxes1;

}

**private** **static** Box[] createBoxes2() {

Box[] boxes2 = **new** Box[3];

boxes2[0] = **new** Box(5,5,5);

boxes2[1] = **new** Box(10,8,8);

boxes2[2] = **new** Box(3,3,3);

**return** boxes2;

}

}

1. The Enhanced For Loop (aka The for-each Loop)

Consider this array:

// Initialize array

**int**[] vals = {7,3,5,2};

Of course, we can iterate over all the values with an *indexed for loop* as shown below:

// Indexed loop

**for**(**int** i=0; i<vals.length; i++) {

**int** x = vals[i];

System.***out***.println(x + " ");

}

When you want to iterate over all the values in an array, the *for-each* loop is simpler. Java calls this the *enhanced for loop*; however, most people refer to it as the *for-each loop.* As an example, the *for-each* loop below provides exactly the same output as the indexed loop above:

// For-each loop

**for**(**int** x : vals) {

System.***out***.println(x + " ");

}

Some notes about the *for-each* loop:

1. We read the statement below, “for each integer *x* in *vals*”

**for**(**int** x : vals)

1. The first time through the loop, *x* is automatically the first value in the array. The second time through, *x* is the second value, *etc.* In other words, Java is picking the appropriate value from the array automatically, and returning it in the variable, *x.* The loop terminates after the last value in the array has been processed.
2. If you need to iterate over one array, from first to last, and you don’t need the index (see examples below), the for-each loop is preferred because it is simpler to read.

If we need the index of a particular element in the array, the for-each loop is not appropriate. For example, suppose you want to write a method to return the location (index) of the smallest element in an array.

**private** **static** **int** locationOfMinimum(**int**[] vals) {

**int** min = vals[0];

**int** locMin = 0;

**for**(**int** i=1; i<vals.length; i++) {

**if**(vals[i]<min) {

min = vals[i];

locMin = i;

}

}

**return** locMin;

}

Note, we could have used a for-each loop, but it is a bit more cumbersome and usually we wouldn’t.

Several other examples of where an indexed loop is preferred:

1. If you need to access every other element in an array (or every third element, *etc.*), then an indexed loop is preferred.

**int**[] vals = {7,3,5,2};

**for**(**int** i=0; i<vals.length; i+=2) {

System.***out***.println(vals[i] + " ");

}

1. If you need to iterate over the elements in a loop backwards.

**int**[] vals = {7,3,5,2};

**for**(**int** i=vals.length-1; i>=0; i--) {

System.***out***.println(vals[i] + " ");

}

1. If you need to iterate over two arrays.

**int**[] xVals = {7,3,5,2};

**int**[] yVals = {1,4,2,6};

**for**(**int** i=0; i<xVals.length; i++) {

**int** x = xVals[i];

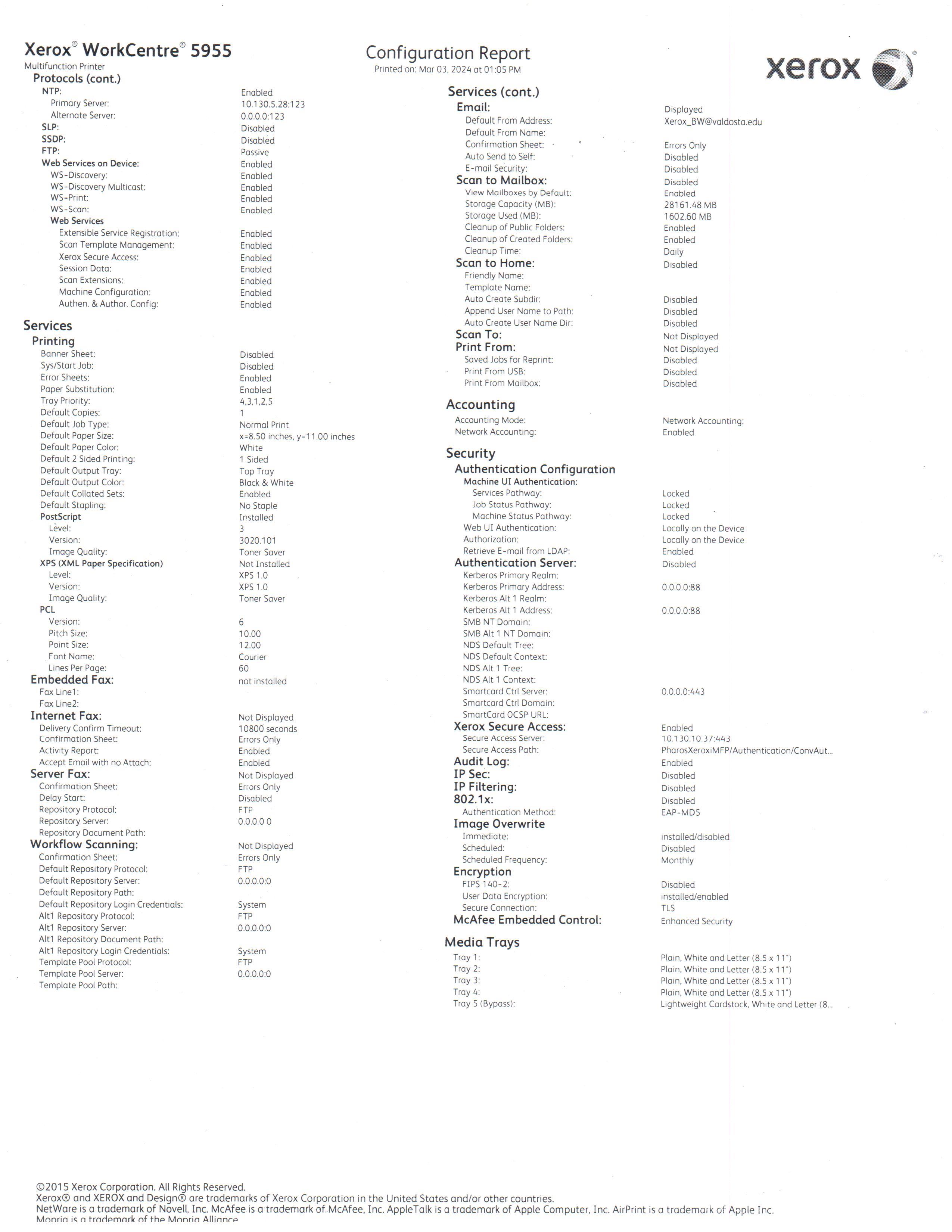
**int** y = yVals[i];

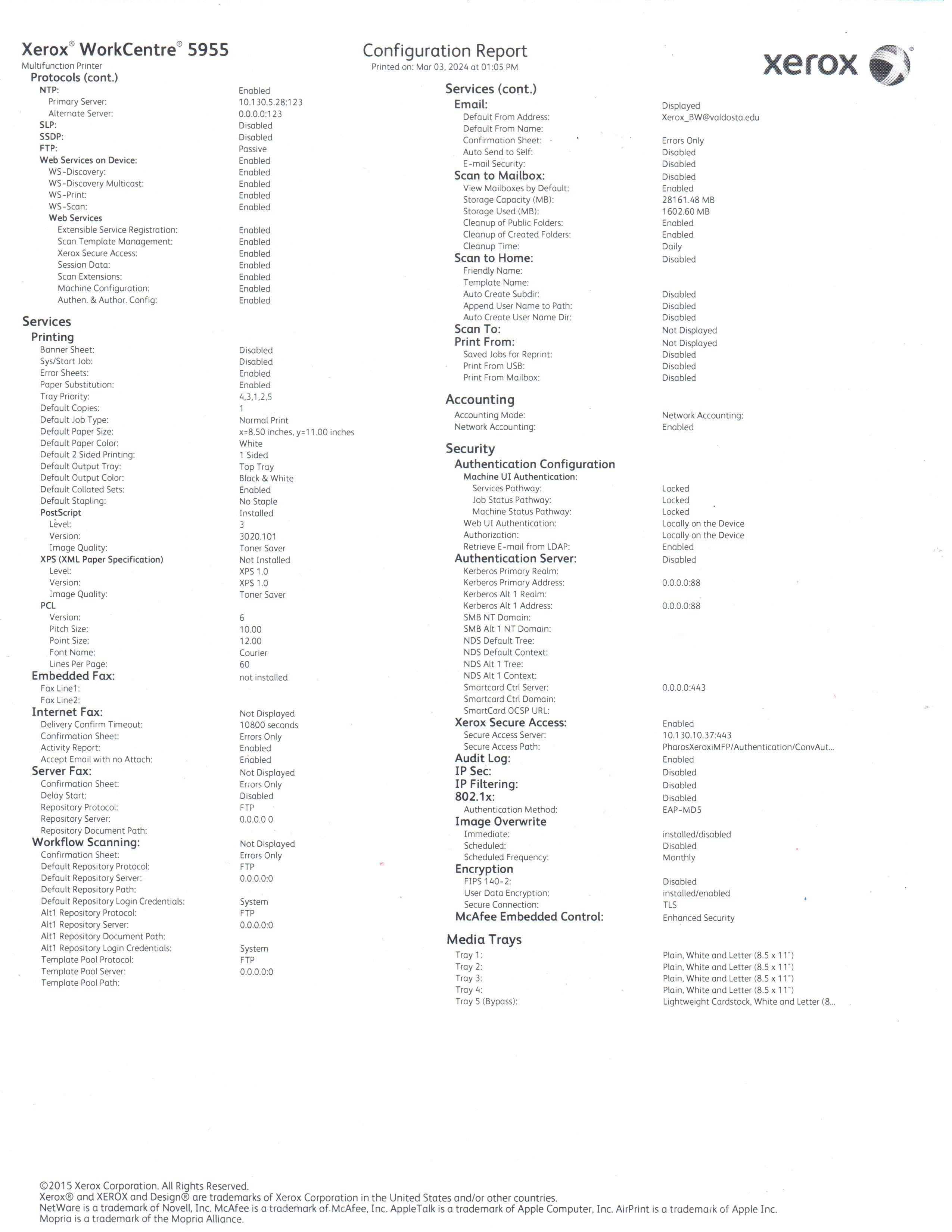
System.***out***.println(x + " " + y + " ");

}

1. State Information for a Network Printer

Below are two pages of a test page generated from a network printer. This is the state information for the printer.





1. The [Java API](https://docs.oracle.com/javase/8/docs/api/) contains more than 4000 classes that we can use to help us solve problems. [↑](#footnote-ref-1)
2. <https://www.cs.colostate.edu/~cs160/.Summer16/resources/Java_printf_method_quick_reference.pdf> [↑](#footnote-ref-2)
3. <https://cseducators.stackexchange.com/questions/6977/real-world-examples-for-the-difference-between-null-and-zero> [↑](#footnote-ref-3)
4. <https://www.rapidtables.com/code/text/ascii-table.html> [↑](#footnote-ref-4)