Chapter 3

Classes and Objects

3.1 Objects as Models

- A program can be thought of as a model of reality, with ***objects*** in the program representing physical objects.
- Properties of objects:
  - ***State*** (information stored within the object)
  - ***Behavior*** (operations that can be performed on the object)

Example 1: Ball-point Pen

- The state of a ball-point pen with a retractable point can be represented by two values:
  - Is the point of the pen exposed?
  - How much ink remains in the pen?
- Operations on a pen include:
  - Press the button at the end of the pen.
  - Move the pen with the point held against a sheet of paper.
  - Replace the pen's cartridge.
  - Determine how much ink remains in the pen.

Example 2: Bank Account

- A state of a bank account includes the account number, the balance, the transactions performed on the account since it was opened, and so forth.
- For simplicity, let's assume that the state of a bank account consists of just the balance in the account.
- Operations on a bank account include:
  - Deposit money into an account.
  - Withdraw money from the account.
  - Check the balance in the account.
  - Close the account.

Example 3: Car

- The state of a car includes the amount of fluids in the car, the state of the tires, and even the condition of each part in the car.
- For programming purposes, we can focus on just a few elements of the state:
  - Is the engine on?
  - How much fuel remains in the car's tank?
- Operations on a car include:
  - Start the engine.
  - Drive a specified distance.

3.2 Representing Objects Within a Program

- In Java, the state of an object is stored in ***instance variables*** (or ***fields***).
- The behavior of an object is represented by ***instance methods.***

Instance Variables

- Some instance variables will store a single value. Others may store entire objects.
- Instance variables needed for a ball-point pen:
  - `pointIsExposed` (`boolean`)
  - `inkRemaining` (`double`)
- Instance variables needed for a bank account:
  - `balance` (`double`)
- Instance variables needed for a car:
  - `engineIsOn` (`boolean`)
  - `fuelRemaining` (`double`)

Instance Methods

- In Java, performing an operation on an object is done by calling one of the instance methods associated with the object.
- An instance method may require arguments when it's called, and it may return a value.
- When asked to perform an operation on an object, an instance method can examine and/or change the values stored in any of the object's instance variables.

Examples of Instance Methods

- Instance methods for ball-point pens:
  - `pressButton`:"Toggles" `pointIsExposed`.
  - `write`: Reduces value of `inkRemaining`.
  - `replaceCartridge`: Restores `inkRemaining` to its maximum value.
  - `checkInkRemaining`: Returns value of `inkRemaining`.
- Instance methods for bank accounts:
  - `deposit`: Adds an amount to `balance`.
  - `withdraw`: Subtracts an amount from `balance`.
  - `getBalance`: Returns value of `balance`.
  - `close`: Stores zero into `balance`.

Examples of Instance Methods

- Instance methods for cars:
  - `startEngine`: Stores `true` into `engineIsOn`.
  - `stopEngine`: Stores `false` into `engineIsOn`.
  - `drive`: Reduces `fuelRemaining` by an amount calculated by dividing the distance traveled by the expected fuel consumption.
  - `addFuel`: Increases `fuelRemaining` by a specified amount.

3.3 Classes

- The instance variables and instance methods that belong to a particular kind of object are grouped together into a ***class.***
- Examples of classes:
  - `BallpointPen`
  - `Account`
  - `Car`

Declaring a Class

- A ***class declaration*** contains declarations of instance variables and instance methods.
- Most class declarations also contain declarations of ***constructors,*** whose job is to initialize objects.
- Form of a class declaration:

```
public class class-name {
    variable-declarations
    constructor-declarations
    method-declarations
}
```

- The order of declarations usually doesn't matter.

Access Modifiers

- The declaration of an instance variable, a constructor, or an instance method usually begins with an ***access modifier*** (`public` or `private`).
- An access modifier determines whether that entity can be accessed by other classes (`public`) or only within the class itself (`private`).
- The most common arrangement is for instance variables to be private and constructors and instance methods to be public.

Declaring Instance Variables

- An instance variable declaration looks the same as the declaration of a variable inside a method, except that an access modifier is usually present:

```
private double balance;
```

- The only access to `balance` will be through the instance methods in the `Account` class.
- The policy of making instance variables private is known as ***information hiding.***

Declaring Instance Methods

- Parts of an instance method declaration:
  - Access modifier
  - Result type. If no value is returned, the result type is `void`.
  - Method name
  - Parameters
  - Body
- Outline of the `deposit` method:

```
public void deposit(double amount) {
   …
}
```

Method Overloading

- Java allows methods to be ***overloaded.*** Overloading occurs when a class contains more than one method with the same name.
- The methods must have different numbers of parameters or there must be some difference in the types of the parameters.
- Overloading is best used for methods that perform essentially the same operation.
- The advantage of overloading: Fewer method names to remember.

Declaring Constructors

- When an object is created, its instance variables are initialized by a constructor.
- A constructor looks like an instance method, except that it has no result type and its name is the same as the name of the class itself.
- A constructor for the `Account` class:

```
public Account(double initialBalance) {
   …
}
```

- A class may have more than one constructor.

Example: An `Account` Class

Account.java

```
public class Account {
  // Instance variables
  private double balance;

  // Constructors
  public Account(double initialBalance) {
    balance = initialBalance;
  }

  public Account() {
    balance = 0.0;
  }
```

```
  // Instance methods
  public void deposit(double amount) {
    balance += amount;
  }

  public void withdraw(double amount) {
    balance -= amount;
  }

  public double getBalance() {
    return balance;
  }

  public void close() {
    balance = 0.0;
  }
}
```

3.4 Creating Objects

- Once a class has been declared, it can be used to create objects (***instances*** of the class).
- Each instance will contain its own copy of the instance variables declared in the class.
- A newly created object can be stored in a variable whose type matches the object's class:

```
Account acct;
```

  Technically, `acct` will store a ***reference*** to an `Account` object, not the object itself.

The `new` Keyword

- The keyword `new`, when placed before a class name, causes an instance of the class to be created.
- A newly created object can be stored in a variable:

```
acct = new Account(1000.00);
```

- The `acct` variable can be declared in the same statement that creates the `Account` object:

```
Account acct = new Account(1000.00);
```

- An object can also be created using the second constructor in the `Account` class:

```
acct = new Account();
```

3.5 Calling Instance Methods

- Once an object has been created, operations can be performed on it by calling the instance methods in the object's class.
- Form of an instance method call:

  object . *method-name* (*arguments*)

  The parentheses are mandatory, even if there are no arguments.

Calling `Account` Instance Methods

- Suppose that `acct` contains an instance of the `Account` class.
- Example calls of `Account` instance methods:

```
acct.deposit(1000.00);
acct.withdraw(500.00);
acct.close();
```

- An object must be specified when an instance method is called, because more than one instance of the class could exist:

```
acct1.deposit(1000.00);
acct2.deposit(1000.00);
```

Using the Value Returned by an Instance Method

- When an instance method returns no result, a call of the method is an entire statement:

```
acct.deposit(1000.00);
```

- When an instance method *does* return a result, that result can be used in a variety of ways.
- One possibility is to store it in a variable:

```
double newBalance = acct.getBalance();
```

- Another possibility is to print it:

```
System.out.println(acct.getBalance());
```

How Instance Methods Work

- Sequence of events when an instance method is called:
  - The program "jumps" to that method.
  - The arguments in the call are copied into the method's corresponding parameters.
  - The method begins executing.
  - When the method is finished, the program "returns" to the point at which the method was called.

3.6 Writing Programs with Multiple Classes

- A program that tests the `Account` class:

TestAccount.java

```
public class TestAccount {
  public static void main(String[] args) {
    Account acct1 = new Account(1000.00);
    System.out.println("Balance in account 1: " +
                       acct1.getBalance());
    acct1.deposit(100.00);
    System.out.println("Balance in account 1: " +
                       acct1.getBalance());
    acct1.withdraw(150.00);
    System.out.println("Balance in account 1: " +
                       acct1.getBalance());
```

```
        acct1.close();
        System.out.println("Balance in account 1: " +
                           acct1.getBalance());

        Account acct2 = new Account();
        System.out.println("Balance in account 2: " +
                           acct2.getBalance());
        acct2.deposit(500.00);
        System.out.println("Balance in account 2: " +
                           acct2.getBalance());
        acct2.withdraw(350.00);
        System.out.println("Balance in account 2: " +
                           acct2.getBalance());
        acct2.close();
        System.out.println("Balance in account 2: " +
                           acct2.getBalance());
    }
}
```

Output of the `TestAccount` program

```
Balance in account 1: 1000.0
Balance in account 1: 1100.0
Balance in account 1: 950.0
Balance in account 1: 0.0
Balance in account 2: 0.0
Balance in account 2: 500.0
Balance in account 2: 150.0
Balance in account 2: 0.0
```

Compiling a Program with Multiple Classes

- The `TestAccount` class, together with the `Account` class, form a complete program.
- If the classes are stored in separate files, they could be compiled using the following commands:

```
javac Account.java
javac TestAccount.java
```

- As an alternative, both files can be compiled with a single command:

```
javac TestAccount.java
```

Compiling a Program with Multiple Classes

- When a file is compiled, the compiler checks whether its dependent classes are up-to-date.
- If the `.java` file containing a dependent class has been modified since the `.class` file was created, `javac` will recompile the `.java` file automatically.
- When `TestAccount.java` is compiled, the `javac` compiler will look for `Account.java` and compile it if necessary.

Executing a Program with Multiple Classes

- Command to execute the `TestAccount` program:

  ```
  java TestAccount
  ```

  The `Account` class is not mentioned.

Using a Single File

- The `Account` and `TestAccount` classes can be put in the same file.
  - The file will need to be named `TestAccount.java`, because `TestAccount` contains the `main` method.
  - The `public` access modifier will have to be removed from the beginning of the `Account` class declaration. (Only one class in a file can be declared `public`.)
- Compiling `TestAccount.java` causes `TestAccount.class` and `Account.class` to be generated.

Using a Single File

- It's often better to put only one class in each file.
- Advantages:
  - Classes are easier to locate.
  - Files are smaller and easier to edit.
  - If a class declaration is changed, only the class itself will have to be recompiled.

Exercise: Write a Program: Account

- Modify the Account.java and TestAccount.java program.
  - Add instance variable to denote accountID, which is 4 digits only and can be randomly generated.
    - double x = Math.random();
  - Test and printout user accountID and balance

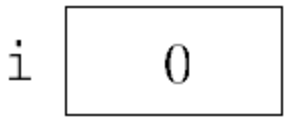
3.7 How Objects Are Stored

- A variable of an ordinary (non-object) type can be visualized as a box:

```
int i;
```

- Assigning a value to the variable changes the value stored in the box:

```
i = 0;
```



Object Variables

- An object variable, on the other hand, doesn't actually store an object. Instead, it will store a ***reference*** to an object.
- An object variable can still be visualized as a box:

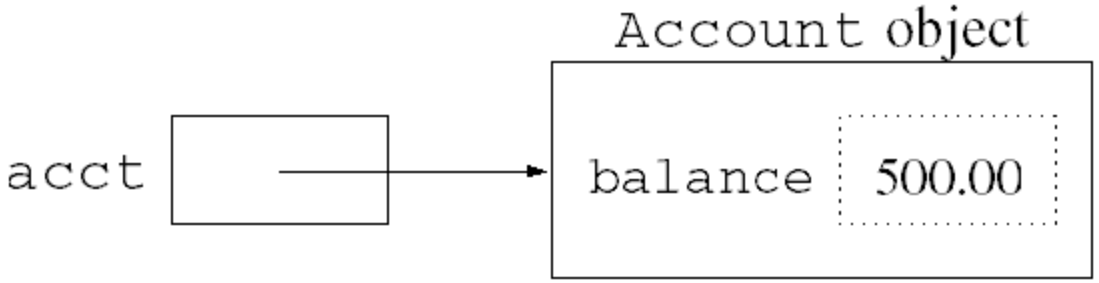
```
Account acct;
```

- Suppose that a new object is stored into `acct`:

```
acct = new Account(500.00);
```

Object Variables

- The `Account` object isn't stored in the `acct` box. Instead, the box contains a reference that "points to" the object:



- In many programming languages, including C++, a variable such as `acct` would be called a ***pointer variable.***

The `null` Keyword

- To indicate that an object variable doesn't currently point to an object, the variable can be assigned the value `null`:

```
acct = null;
```

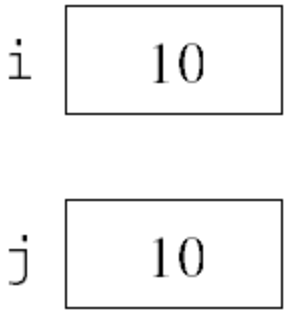
- When an object variable stores `null`, it's illegal to use the variable to call an instance method.
- If `acct` has the value `null`, executing the following statement will cause a run-time error (`NullPointerException`):

```
acct.deposit(500.00);
```

Object Assignment

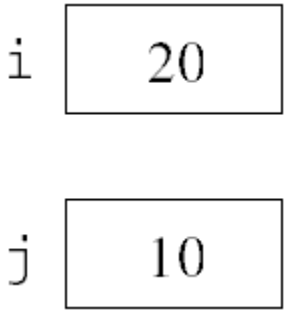
- If `i` has the value 10, assigning `i` to `j` gives `j` the value 10 as well:

```
j = i;
```



- Changing the value of `i` has no effect on `j`:

```
i = 20;
```

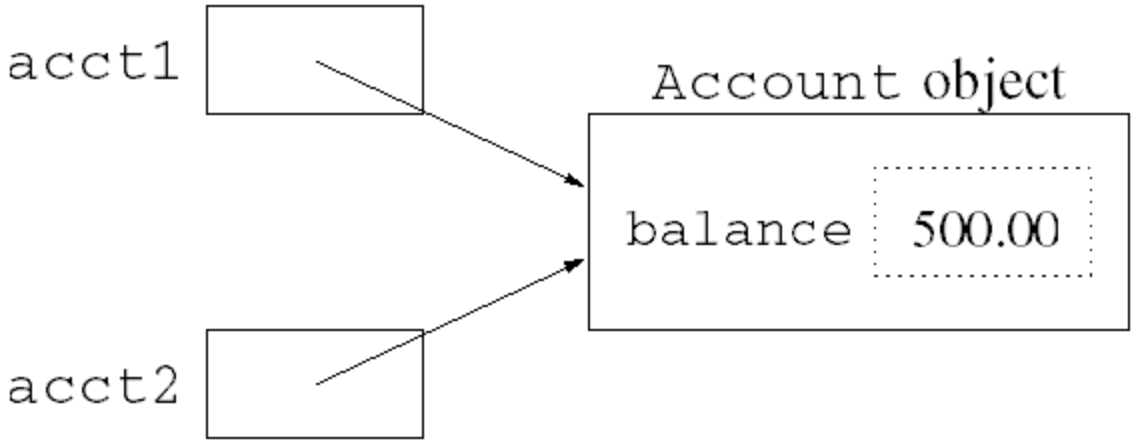


- Assignment of objects doesn't work the same way.

Object Assignment

- Assume that `acct1` contains a reference to an `Account` object with a balance of $500.
- Assigning `acct1` to `acct2` causes `acct2` to refer to the same object as `acct1`:

```
acct2 = acct1;
```



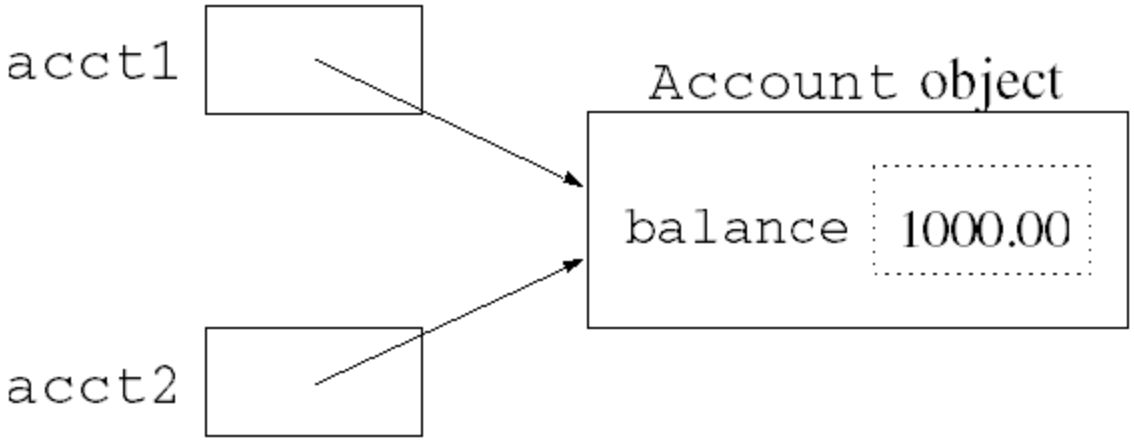
- `acct1` and `acct2` are said to be ***aliases,*** because both represent the same object.

Object Assignment

- An operation that changes the `acct1` object will also change the `acct2` object, and vice-versa.
- The statement

  ```
  acct1.deposit(500.00);
  ```

  will change the balance of `acct2` to $1000.00:



Cloning

- Some classes allow the creation of a new object that's identical to an existing object.
- The new object is said to be a ***clone*** of the old one.
- Clones are created by calling the `clone` method.

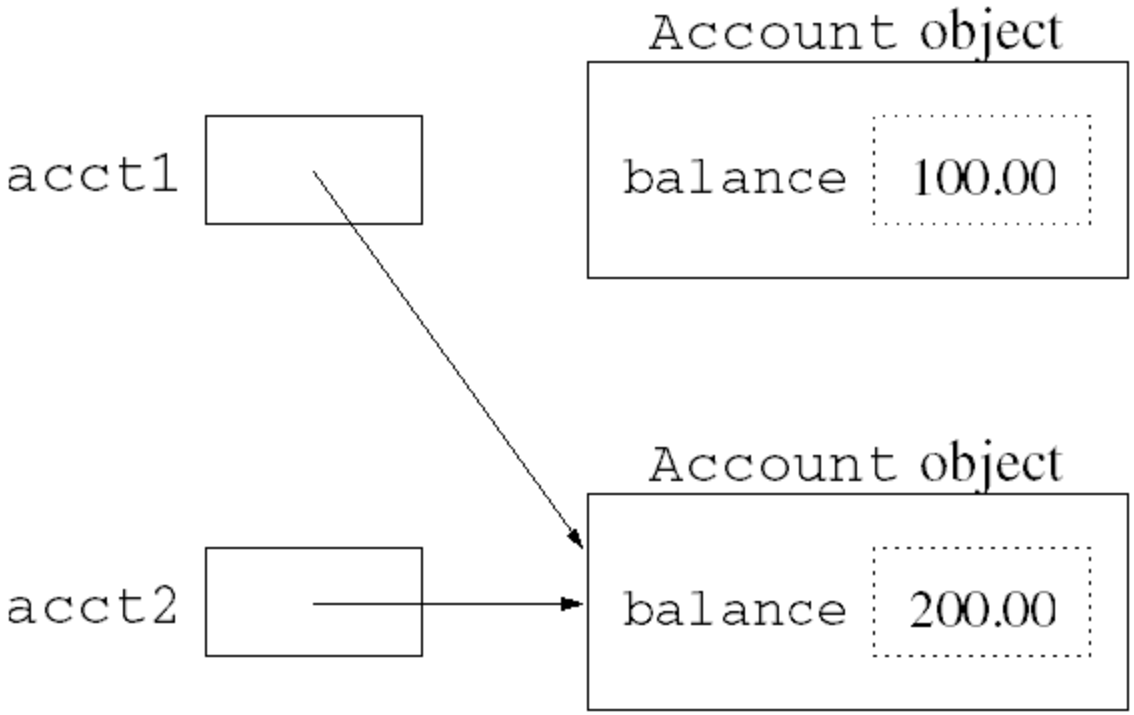
Garbage

- Objects can become "orphaned" during program execution.
- Consider the following example:

```
acct1 = new Account(100.00);
acct2 = new Account(200.00);
acct1 = acct2;
```

- After these assignments, the object that `acct1` previously referred to is lost. We say that it is ***garbage.***

Garbage



Garbage Collection

- Java provides automatic ***garbage collection:*** as a Java program runs, a software component known as the ***garbage collector*** watches for garbage and periodically "collects" it.
- The recycled memory can be used for the creation of new objects.
- Garbage collection normally takes place when the program isn't doing any other useful activity.
- Java is the first widely used programming language to incorporate garbage collection .

Memory Leaks

- Other popular languages rely on the program to explicitly release memory that's no longer needed.
- This practice is potentially more efficient, but it's also error-prone.
- Failing to recover garbage causes available memory to decrease (a ***memory leak***).
- After a period of time, a program with a memory leak may run out of memory entirely.
- Releasing memory prematurely is even worse, often causing programs to crash.

Exercise: Write a Program

- Test the following code and explain

```
acct1 = new Account(100.00);
acct2 = new Account(200.00);
acct1 = acct2;
acct2.deposit (500);
acct3 = new Account (200);
acct3.withdraw(300);
```

What are the balances of acct1,acct2, acct3? why?

3.8 Developing a `Fraction` Class

- Fractions can be thought of as objects, so it's not hard to develop a `Fraction` class.
- A `Fraction` object will need to store a numerator and a denominator. Both are integers.
- There are many potential operations on fractions, including adding, subtracting, multiplying, and dividing.

A First Attempt

- A first attempt at writing the `Fraction` class:

```
public class Fraction {
   private int numerator;
   private int denominator;

   public Fraction(int num, int denom) {
      numerator = num;
      denominator = denom;
   }

   // Methods will go here
}
```

- A `Fraction` object will be created as follows:

```
Fraction f = new Fraction(4, 8);
```

Getters and Setters

- The `Fraction` class will need methods named `getNumerator` and `getDenominator`:

```
public int getNumerator() {
  return numerator;
}

public int getDenominator() {
  return denominator;
}
```

- An instance method that does nothing but return the value of an instance variable is said to be an ***accessor*** (or a ***getter***).

Getters and Setters

- By convention, names of getters start with the word `get`.
- Sample calls of `getNumerator` and `getDenominator`:

```
int num = f.getNumerator();
int denom = f.getDenominator();
```

- An instance method that stores its parameter into an instance variable is said to be a ***mutator*** (or ***setter***).
- Names of setters begin with the word `set`.

Getters and Setters

- Potential setters for the `Fraction` class:

```
public void setNumerator(int num) {
  numerator = num;
}

public void setDenominator(int denom) {
  denominator = denom;
}
```

- Sample calls of `setNumerator` and `setDenominator`:

```
f.setNumerator(5);
f.setDenominator(6);
```

Immutable Objects

- Setters can be useful, because they allow us to change data stored in private variables.
- In some cases, however, we may not want to allow changes to an object's instance variables.
- Such an object is said to be ***immutable*** (unchangeable).
- The advantage of making objects immutable is that they can be shared without problems.
- Some of the classes in the Java API have this property, including the `String` class.

Writing the `add` Method

- A method that adds `Fraction` objects `f1` and `f2` would need to be called in the following way:

```
Fraction f3 = f1.add(f2);
```

- `add` would have the following appearance:

```
public Fraction add(Fraction f) {
   …
}
```

The parameter `f` represents the second of the two fractions to be added.

Writing the `add` Method

- A first attempt at writing the `add` method:

```
public Fraction add(Fraction f) {
  int num = numerator * f.getDenominator() +
            f.getNumerator() * denominator;
  int denom = denominator * f.getDenominator();
  Fraction result = new Fraction(num, denom);
  return result;
}
```

- `numerator` and `denominator` refer to the numerator and denominator of the `Fraction` object that's calling `add`.

Writing the `add` Method

- The `add` method can be shortened slightly by combining the constructor call with the `return` statement:

```
public Fraction add(Fraction f) {
  int num = numerator * f.getDenominator() +
            f.getNumerator() * denominator;
  int denom = denominator * f.getDenominator();
  return new Fraction(num, denom);
}
```

Writing the `add` Method

- The `add` method can be further simplified by having it access `f`'s `numerator` and `denominator` variables directly:

```
public Fraction add(Fraction f) {
  int num = numerator * f.denominator +
            f.numerator * denominator;
  int denom = denominator * f.denominator;
  return new Fraction(num, denom);
}
```

- Instance variables are accessed using a dot, just as instance methods are called using a dot.

Adding a `toString` Method

- The value stored in a `Fraction` object named `f` could be printed in the following way:

```
System.out.println(f.getNumerator() + "/" +
                   f.getDenominator());
```

- The following method makes it easier to print fractions:

```
public String toString() {
  return numerator + "/" + denominator;
}
```

- In Java, the name `toString` is used for a method that returns the contents of an object as a string.

Adding a `toString` Method

- The `toString` method makes it easier to display the value stored in a `Fraction` object:

```
System.out.println(f.toString());
```

- The statement can be shortened even further:

```
System.out.println(f);
```

When given an object as its argument, `System.out.println` will automatically call the object's `toString` method.

Exercise: Write a Program

- Add the **`subtract`** Method for class Fraction
- Test the new Fraction and output results

```
5/6 - 2/6
5/6 - 1/3
```

3.9 Java's `String` Class

- The Java API provides a huge number of prewritten classes. Of these, the `String` class is probably the most important.
- Instances of the `String` class represent strings of characters.
- The `String` class belongs to a package named `java.lang`.
- The `java.lang` package is automatically imported into every program. (No other package has this property.)

Creating Strings

- In Java, every string of characters, such as `"abc"`, is an instance of the `String` class.
- `String` variables can be assigned `String` objects as their values:

```
String str1, str2;
```

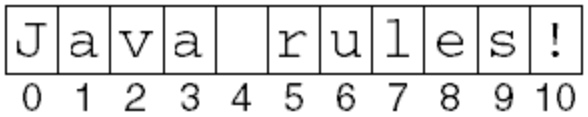
- `String` is the only class whose instances can be created without the word `new`:

```
str1 = "abc";
```

This is an example of ***magic.***

Visualizing a String

- A `String` object can be visualized as a series of characters, with each character identified by its position.
- The first character is located at position 0.
- A visual representation of the string `"Java rules!"`:



Common String Methods

- The `String` class has a large number of instance methods.
- Assume that the following variable declarations are in effect:

```
String str1 = "Fat cat", str2;
char ch;
int index;
```

- The `charAt` method returns the character stored at a specific position in a string:

```
ch = str1.charAt(0);  // Value of ch is now 'F'
ch = str1.charAt(6);  // Value of ch is now 't'
```

Common String Methods

- One version of the `indexOf` method searches for a string (the "search key") within a larger string, starting at the beginning of the larger string.
- *Example:* Locating the string `"at"` within `str1`:

```
index = str1.indexOf("at");
```

  After this assignment, index will have the value 1.
- If `"at"` had not been found anywhere in `str1`, `indexOf` would have returned –1.

Common String Methods

- The other version of `indexOf` begins the search at a specified position, rather than starting at position 0.
- This version is particularly useful for repeating a previous search to find another occurrence of the search key.
- *Example:* Finding the second occurrence of `"at"` in `str1`:

```
index = str1.indexOf("at", index + 1);
```

`index` will be assigned the value 5.

Common String Methods

- `lastIndexOf` is similar to `indexOf`, except that searches proceed backwards, starting from the end of the string.
- *Example:* Finding the last occurrence of `"at"` in `str1`:

```
index = str1.lastIndexOf("at");
```

The value of `index` after the assignment will be 5.

Common String Methods

- The second version of `lastIndexOf` begins the search at a specified position.
- *Example:* Finding the next-to-last occurrence of `"at"`:

```
index = str1.lastIndexOf("at", index - 1);
```

  The value of `index` after the assignment will be 1.

- The `String` class has additional versions of `indexOf` and `lastIndexOf`, whose first argument is a single character rather than a string.

Common String Methods

- The `length` method returns the number of characters in a string.
- For example, `str1.length()` returns the length of `str1`, which is 7.
- The `substring` method returns a ***substring:*** a series of consecutive characters within a string.
- One version of `substring` selects a portion of a string beginning at a specified position:

```
str2 = str1.substring(4);
```

After the assignment, `str2` will have the value `"cat"`.

Common String Methods

- The other version of `substring` accepts two arguments:
  - The position of the first character to include in the substring
  - The position of the first character *after* the end of the substring
- Example:

```
str2 = str1.substring(0, 3);
```

After the assignment, `str2` will have the value `"Fat"`.

Common String Methods

- `toLowerCase` and `toUpperCase` will convert the letters in a string to lowercase or uppercase.
- After the assignment

  ```
  str2 = str1.toLowerCase();
  ```

  the value of `str2` is `"fat cat"`.
- After the assignment

  ```
  str2 = str1.toUpperCase();
  ```

  the value of `str2` is `"FAT CAT"`.
- Characters other than letters aren't changed by `toLowerCase` and `toUpperCase`.

Common String Methods

- The `trim` method removes spaces (and other invisible characters) from both ends of a string.
- After the assignments

```
str1 = "  How  now,  brown  cow?  ";
str2 = str1.trim();
```

the value of `str2` will be

```
"How  now,  brown  cow?"
```

Chaining Calls of Instance Methods

- When an instance method returns an object, that object can be used to call another instance method.
- For example, the statements

```
str2 = str1.trim();
str2 = str2.toLowerCase();
```

can be combined into a single statement:

```
str2 = str1.trim().toLowerCase();
```

Using + to Concatenate Strings

- One of the most common string operations is ***concatenation:*** joining two strings together to form a single string.
- The `String` class provides a `concat` method that performs concatenation, but it's rarely used.
- Concatenation is so common that Java allows the use of the plus sign (+) to concatenate strings:

```
str2 = str1 + "s";
```

`str2` now contains the string `"Fat cats"`.

Using + to Concatenate Strings

- The + operator works even if one of the operands isn't a `String` object. The non-`String` operand is converted to string form automatically:

```
System.out.println("Celsius equivalent: " +
                   celsius);
```

Using + to Concatenate Strings

- If the + operator is used to combine a string with any other kind of object, the object's `toString` method is called.
- The statement

```
System.out.println("Value of fraction: " + f);
```

has the same effect as

```
System.out.println("Value of fraction: " +
                   f.toString());
```

Using + to Concatenate Strings

- In order for the + operator to mean string concatenation, at least one of its two operands must be a string:

```
System.out.println("Java" + 1 + 2);
  // Prints "Java12"
System.out.println(1 + 2 + "Java");
  // Prints "3Java"
```

Using + to Concatenate Strings

- The + operator is useful for breaking up long strings into smaller chunks:

```
System.out.println(
   "Bothered by unsightly white space? " +
   "Remove it quickly and\neasily with " +
   "the new, improved trim method!");
```

Using + to Concatenate Strings

- The += operator can be used to add characters to the end of a string:

```
String str = "The quick brown fox ";
str += "jumped over ";
str += "the lazy dog.";
```

  The final value of `str` will be `"The quick brown fox jumped over the lazy dog."`

- Concatenating a number with an empty string will convert the number to string form. For example, if `i` contains `37`, then `i + ""` is the string `"37"`.

Exercise: Write a Program

- Modify the Account class
  - Add *username* as an instance variable
  - Add setUserName method
    - acct1.setUserName("Tom White");
  - Add getName method
  - Test it and print out all the information of a bank account in the following format

```
username(accountID) has balance in the bank
e.g.,
Tom Whiten (3420) has $100.00 in the bank
```