Think Java

CHAPTER 11: DESIGN CLASSES

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Designing classes

Designing Classes

1. In this chapter, you will learn to design classes that represent useful objects. Think of a class like a **blueprint** for a house: you can use the same blueprint to build any number of houses.
2. Beginners often confuse the difference between classes and objects. Here are the main ideas:
   - A class definition is a **template** for objects: it specifies what attributes the objects have and what methods can operate on them.

Designing Classes

- Every object belongs to some object type; that is, it is an instance of some class.
- The new operator instantiates objects, that is, it creates new instances of a class.
- The design of a class (what methods it has) determines whether the objects are **mutable** or **immutable**.

LECTURE 1

The Time class

The Time class

- One common reason to define a new class is to encapsulate related data in an object that can be treated as a single unit. That way, we can use objects as parameters and return values, rather than passing and returning multiple values. We have already seen two types that encapsulate data in this way: **Point** and **Rectangle**.
- Another example, which we will implement ourselves, is Time, which represents a time of day. The data encapsulated in a Time object include an hour, a minute, and a number of seconds. Because every Time object contains these data, we define attributes to hold them.
- Attributes are also called **instance variables**, because each instance has its own variables (as opposed to class variables, coming up in Section 12.3).

The Time class

- The first step is to decide what type each variable should be. It seems clear that hour and minute should be integers. Just to keep things interesting, let's make second a double.
- Instance variables are declared at the beginning of the class definition, outside of any method. By itself, this code fragment is a legal class definition:

```
public class Time {
    private int hour;
    private int minute;
    private double second;
}
```

Information Hiding

- The **Time** class is public, which means that it can be used in other classes. But the instance variables are private, which means they can only be accessed from inside the **Time** class.
- If you try to read or write them from another class, you will get a compiler error.
- Private instance variables help keep classes isolated from each other so that changes in one class won't require changes in other classes.
- It also simplifies what other programmers need to understand in order to use your classes. This kind of isolation is called **information hiding**.

Exercise 1:

Write a Date0 Class with the following data fields:

Year, Month, and Day.

LECTURE 3

Constructors

Constructor

After declaring instance variables, the next step is to define a **constructor**, which is a special method that initializes the object. The syntax for constructors is similar to that of other methods, except:

- The name of the constructor is the same as the name of the class.
- Constructors have no return type (and no return value).
- The keyword static is omitted.

Constructor

- Here is an example constructor for the Time class:

```
public Time() {
   this.hour = 0;
   this.minute = 0;
   this.second = 0.0;
}
```

- This constructor does not take any arguments. Each line initializes an instance variable to zero (which in this example means midnight).

this keyword

- The name this is a keyword that refers to the object we are creating. You can use this the same way you use the name of any other object. For example, you can read and write the instance variables of this, and you can pass this as an argument to other methods. But you do not declare this, and you can't make an assignment to it.
- A common error when writing constructors is to put a return statement at the end. Like void methods, constructors do not return values.

new Operator

- To create a Time object, you must use the new operator:

```
public static void main(String[] args) {
    Time time = new Time();
}
```

- When you use new, Java creates the object and invokes your constructor to initialize the instance variables. When the constructor is done, new returns a reference to the new object. In this example, the reference gets assigned to the variable time, which has type Time. Figure 11.1 shows the result.

Constructor

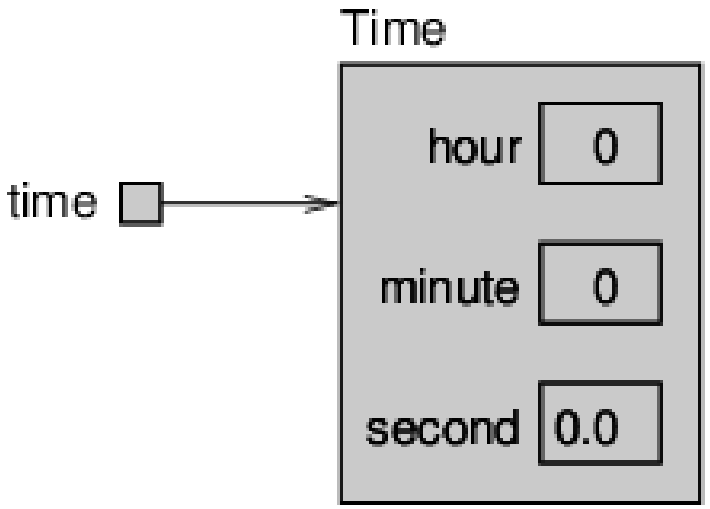


Figure 11.1: Memory diagram of a Time object.

- Beginners sometimes make the mistake of using new in the constructor. Doing so causes an infinite recursion, since new invokes the same constructor, which uses new again, which invokes the constructor again, and so on.

```
public Time() {
    new Time(); // StackOverflowError
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
}
```

Exercise 2:

Add a Default constructor to Date0 Class

Constructor without parameters.

LECTURE 4

More constructors

More constructors

- Like other methods, constructors can be overloaded, which means you can provide **multiple constructors** with different parameters. Java knows which constructor to invoke by matching the arguments you provide with the parameters of the constructor.
- It is common to provide a constructor that takes **no** arguments, like the previous one, and a "**value constructor**", like this one:

```
public Time(int hour, int minute, double second){
  this.hour = hour;
  this.minute = minute;
  this.second = second;
}
```

Overloading

- All this constructor does is copy values from the parameters to the instance variables. In this example, the names and types of the parameters are the same as the instance variables. As a result, the parameters **shadow** (or hide) the instance variables, so the keyword this is necessary to tell them apart. Parameters don't have to use the same names, but that's a common style.
- To invoke this second constructor, you have to provide arguments after the new operator. This example creates a Time object that represents a fraction of a second before noon:

```
Time time = new Time(11, 59, 59.9);
```

Overloading

- Overloading constructors provides the flexibility to create an object first and then fill in the attributes, or collect all the information before creating the object itself.
- Once you get the hang of it, writing constructors gets boring. You can write them quickly just by looking at the list of instance variables. In fact, some IDEs can generate them for you.

Demo Program: Time0.java

- Data fields
- Constructor
- Overloading of Class constructor: Showing the design of constructors (with and without parameters.)

Exercise 3:

Add an overloading constructor with parameters to Date0 Class

Constructor with parameters.

LECTURE 5

Getters and setters

Getters and Setters

- Recall that the instance variables of **Time** are private. We can access them from within the **Time** class, but if we try to access them from another class, the compiler reports an error.
- A class that uses objects defined in another class is called a **client**. For example, here is a new class called **TimeClient**.

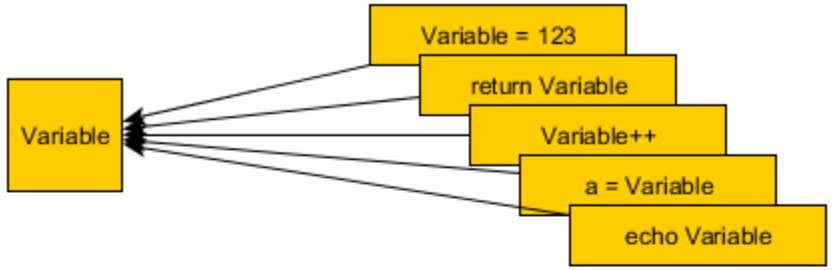
```
public class TimeClient {
  public static void main(String[] args) {
    Time time = new Time(11, 59, 59.9);
    System.out.println(time.hour); // compiler error
  }
}
```

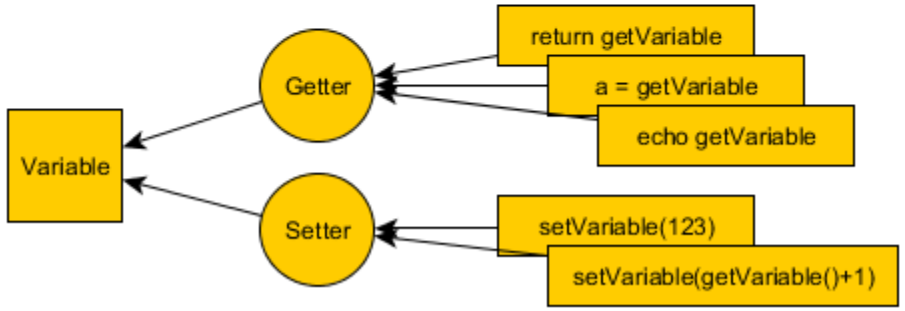
Getters and Setters

- If you try to compile this code, you will get an error message like "hour has private access in Time". There are three ways to solve this problem:
  - We could make the instance variables `public`.
  - We could provide methods to access the instance variables.
  - We could decide that it's not a problem, and refuse to let other classes access the instance variables.
- The first choice is appealing because it's simple. But here is the problem: when Class *A* accesses the instance variables of Class *B* directly, *A* becomes dependent on *B*. If anything in *B* changes later, it is likely that *A* will have to change, too.

Getters and Setters

- But if A only uses methods to interact with B, A and B are less dependent, which means that we can make changes in B without affecting A (as long as we don't change the method parameters).





Getters

- So if we decide that **TimeClient** should be able to read the instance variables of Time, we should provide methods to do it:

```
public int getHour() {
   return this.hour;
}
public int getMinute() {
   return this.minute;
}
public double getSecond() {
   return this.second;
}
```

- Methods like these are formally called "**accessors**", but more commonly referred to as getters. By convention, the method that gets a variable named something is called **getSomething**.

Setters

- If we decide that **TimeClient** should also be able to modify the instance variables of **Time**, we can provide methods to do that, too:

```
public void setHour(int hour) {
this.hour = hour;
}
public void setMinute(int minute) {
 this.minute = minute;
}
public void setSecond(double second)
{
 this.second = second;
}
```

- These methods are formally called “mutators”, but more commonly known as setters. The naming convention is similar; the method that sets something is usually called setSomething.

Automatic generation of getters and setters

Writing getters and setters can get boring, but many IDEs can generate them for you based on the instance variables.

Exercise 4:

Write the getters and setters for Date Class

Date4 Class

LECTURE 6

Displaying objects

Displaying objects

- If you create a Time object and display it with println:

```
public static void main(String[] args) {
 Time time = new Time(11, 59, 59.9);
 System.out.println(time);
}
```

- The output will look something like:

```
Time@80cc7c0
```

- When Java displays the value of an object type, it displays the name of the type and the address of the object (in hexadecimal). This address can be useful for debugging, if you want to keep track of individual objects.

Displaying objects

- To display **Time** objects in a way that is more meaningful to users, you could write a method to display the hour, minute, and second. Using **printTime** in Section 5.4 as a starting point, we could write:

```
public static void printTime(Time t) {
   System.out.print(t.hour);
   System.out.print(":");
   System.out.print(t.minute);
   System.out.print(":");
   System.out.println(t.second);
}
```

Displaying objects

- The output of this method, given the time object from the first example, would be 11:59:59.9. We can use printf to make the code more concise:

```
public static void printTime(Time t) {
 System.out.printf("%02d:%02d:%04.1f\n", t.hour, t.minute, t.second);
}
```

- As a reminder, you need to use \%d with integers and \%f with floating-point numbers.
- The 02 option means “total width 2, with leading zeros if necessary”, and the 04.1 option means “total width 4, one digit after the decimal point, leading zeros if necessary”.
- The output is the same: 11:59:59.9.

Exercise 5:

Try to print out the data fields of a Date object.

YYYY/MM/DD format

Date5 Class

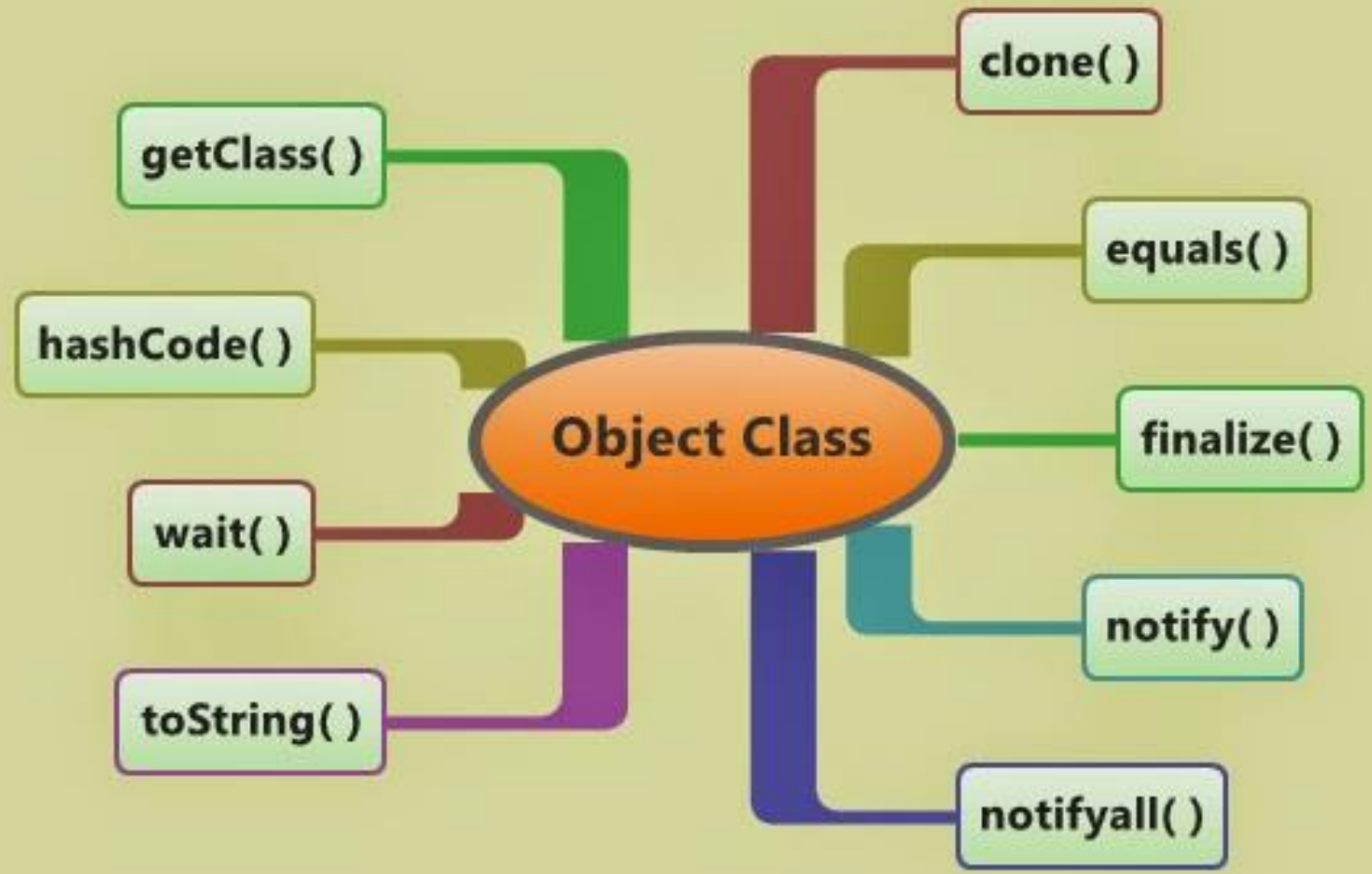
LECTURE 6

Object Class

The `Object` Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
  - even if a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
  - the `Object` class is therefore the ultimate root of all class hierarchies

- The `Object` class contains a few useful methods, which are inherited by all classes
  - `toString()`
  - `equals()`
  - `clone()`


Object Class
clone( )
equals( )
finalize( )
notify( )
notifyall( )
getClass( )
hashCode( )
wait( )
toString( )

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Two most important methods

- toString(): convert the object to printable format.
- equals(): check equality

Important Interface

- Comparable: compareTo() – allowing sorting algorithm.
- Iterable: hasNext(), next(), remove() – allowing element access for data container objects.

LECTURE 7

The toString method

toString() Method

- Every object type has a method called toString that returns a string representation of the object. When you display an object using print or println, Java invokes the object's toString method.
- By default it simply displays the type of the object and its address, but you can override this behavior by providing your own toString method. For example, here is a toString method for Time:

```
public String toString() {
    return String.format("%02d:%02d:%04.1f\n",
                         this.hour, this.minute, this.second);
}
```

toString() Method

- The definition does not have the keyword static, because it is not a static method. It is an **instance method**, so called because when you invoke it, you invoke it on an instance of the class (**Time** in this case). Instance methods are sometimes called "non-static"; you might see this term in an error message.
- The body of the method is similar to **printTime** in the previous section, with two changes:
  - Inside the method, we use this to refer to the current instance; that is, the object the method is invoked on.
  - Instead of **printf**, it uses **String.format**, which returns a formatted **String** rather than displaying it.

toString() Method

Now you can call **toString** directly:

```
Time time = new Time(11, 59, 59.9);
String s = time.toString();
```

Or you can invoke it indirectly through println:

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

In this example, this in toString refers to the same object as time. The return value is "11:59:59.9".

Exercise 6:

Write the toString() method for Date class

Date6 Class

LECTURE 8

The equals method

Equality Check

- We have seen two ways to check whether values are equal: the == operator and the **equals** method. With objects you can use either one, but they are not the same.
  - The == operator checks whether two references are **identical**; that is, whether they refer to the same object.
  - The **equals** method checks whether two objects are **equivalent**; that is, whether they have the same values.
- The definition of identity is always the same, so the == operator always does the same thing.
- But the definition of equivalence is different for different objects, so objects can define their own equals methods.

Equality Check

Consider the following variables and the memory diagram in Figure 11.2.

```
Time time1 = new Time(9, 30, 0.0);
Time time2 = time1;
Time time3 = new Time(9, 30, 0.0);
```

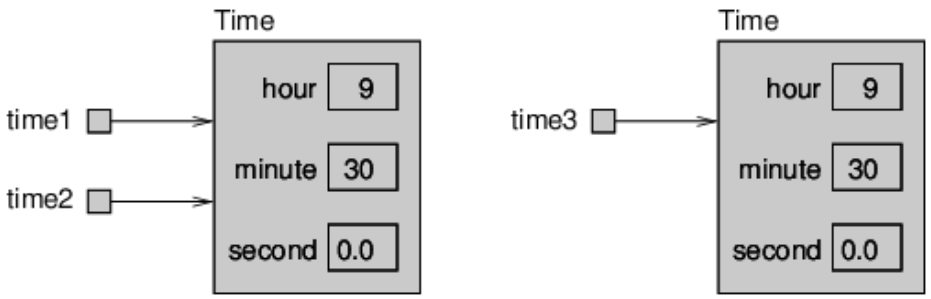


Figure 11.2: Memory diagram of three Time variables.

- The assignment operator copies references, so time1 and time2 refer to the same object.
- Because they are identical, time1 == time2 is true. But time1 and time3 refer to two different objects. Because they are not identical, time1 == time3 is false.

equals() Method

- By default, the **equals** method does the same thing as ==. For Time objects, that's probably not what we want. For example, time1 and time3 represent the same time of day, so we should consider them equivalent.
- We can provide an **equals** method that implements this idea:

```
public boolean equals(Time that) {
  return this.hour == that.hour &&
         this.minute == that.minute &&
         this.second == that.second;
}
```

equals() Method

- **equals** is an instance method, so it doesn't have the keyword static. It uses this to refer to current object, and that to refer to the other. The parameter named that is not a keyword and could have a different name, but it improves readability. We can invoke equals as follows:

```
time1.equals(time3);
```

- Inside the **equals** method, this refers to the same object as time1, and that refers to the same object as time3. Since their instance variables are equal, the result is true.

equals() Method

- Many objects have a similar notion of equivalence; that is, two objects are considered equal if their instance variables are equal. But other definitions are possible. You could, for example, allow a **Time** object and a **String** object to be considered equal if they represent the same time.

```
public boolean equals(String str) {
   return str.equals(this.toString());
}
```

- The **equals** method is now overloaded. If we invoke **time1.equals(time3)**, the first method will be used; time1.equals("09:30:00.0") uses the second.

Exercise 7:

Write the equals() method for Date Class

Date7 class

LECTURE 9

Member Methods: Adding times

Operation between Objects - add

- Suppose you are going to a movie that starts at 18:50 (or 6:50 PM), and the running time is 2 hours 16 minutes. What time does the movie end?
- We'll use Time objects to figure it out.

```
Time startTime   = new Time(18, 50, 0.0);
Time runningTime = new Time(2, 16, 0.0);
```

- Here are two ways we could "**add**" the **Time** objects:
  - We could write a static method that takes two **Time** objects as parameters (not object-oriented).
  - We could write an instance method that gets invoked on one object and takes the other as a parameter

add Method

- To demonstrate the difference, we'll do both. Here is a simple version that uses the static approach:

```
public static Time add(Time t1, Time t2) {
 Time sum = new Time();
 sum.hour = t1.hour + t2.hour;
 sum.minute = t1.minute + t2.minute;
 sum.second = t1.second + t2.second;
 return sum;
}
```

add Method

- And here's how we would invoke the static method:

```
Time endTime = Time.add(startTime, runningTime);
```

- On the other hand, here's what it looks like as an instance method:

```
public Time add(Time t2) {
   Time sum = new Time();
   sum.hour = this.hour + t2.hour;
   sum.minute = this.minute + t2.minute;
   sum.second = this.second + t2.second;
   return sum;
}
```

Instance add method

- The changes are:
  - We removed the keyword static.
  - We removed the first parameter.
  - We replaced t1 with this.
- And here's how we would invoke the instance method:

```
Time endTime = startTime.add(runningTime);
```

- That's all there is to it. Static methods and instance methods do the same thing, and you can convert from one to the other with just a few changes.

add Method

- There's only one problem: the addition code itself is not correct. For this example, it returns 20:66, which is not a valid time.
- If second exceeds 59, we have to "carry" into the minutes column, and if minute exceeds 59, we have to carry into hour. Here is a better version of add:

```
public Time add(Time t2) {
  Time sum = new Time();
  sum.hour = this.hour + t2.hour;
  sum.minute = this.minute + t2.minute;
  sum.second = this.second + t2.second;
  if (sum.second >= 60.0) { sum.second -= 60.0; sum.minute += 1; }
  if (sum.minute >= 60) { sum.minute -= 60; sum.hour += 1; }
  return sum;
}
```

add method

It’s still possible that **hour** may exceed 23, but there’s no days attribute to carry into. In that case, **sum.hour -= 24** would yield the correct result.

Destructive add and Non-destructive add

Destructive add is also called accumulator

Non-destructive add is called addition

Non-Destructive Addition:

```
public Item add(Item other){
    Item result = this.data + other.data;
    return result;
}
```

Destructive Accumulation:

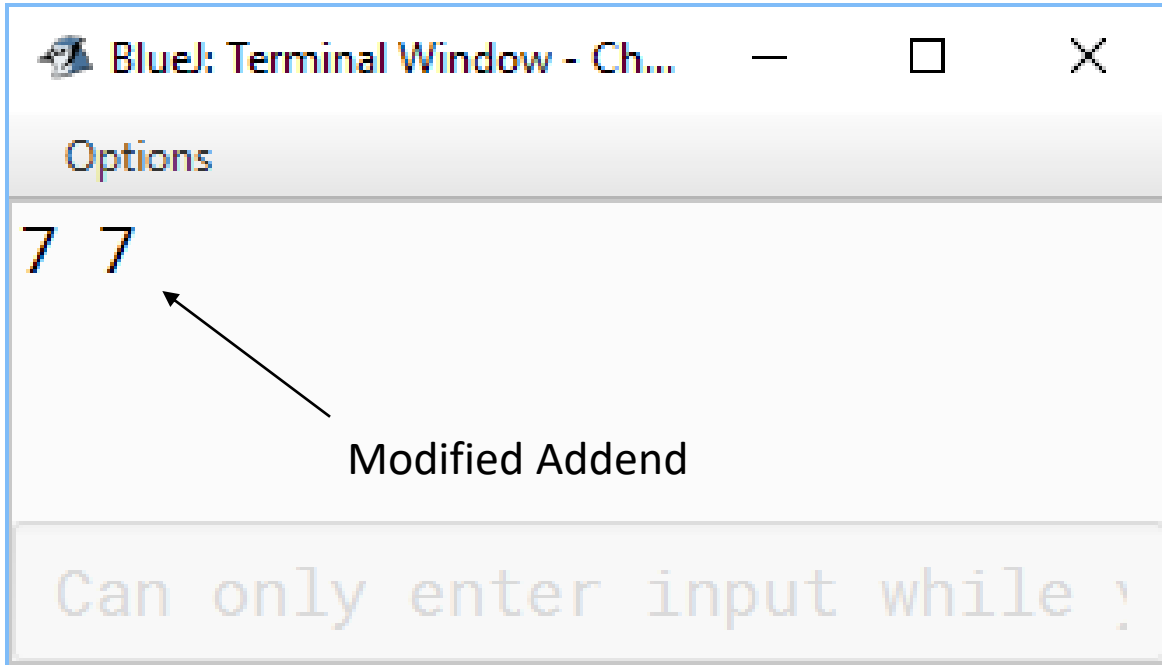
```
public void add(Item other){
    this.data = this.data + other.data;
}
```

Demo Program: Accumulator.java and Addition.java

Go BlueJ!!!

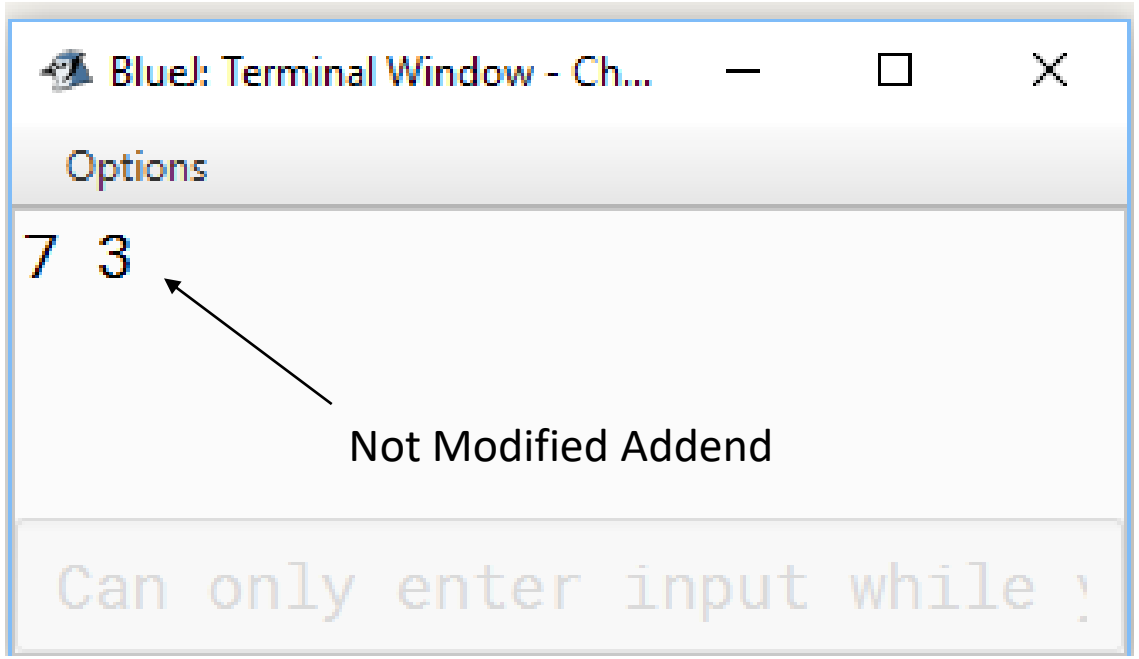
Destructive Addition (Accumulation)

```
public class Accumulator{
    Integer data;
    Accumulator(Integer d){
       data = d;
    }
    public Integer get(){
       return data;
    }
    public void set(Integer i){
        this.data = i;
    }
    public Integer accumulate(Integer other){
       this.data = this.data + (int) other;
       return this.data;
    }
    public Integer accumulate(Accumulator other){
       this.data = this.data + other.data;
       return this.data;
    }
    public String toString(){
      return data.toString();
    }

    public static void main(String[] args){
       Accumulator a = new Accumulator(3);
       Accumulator b = new Accumulator(4);
       System.out.println(a.accumulate(b)+" "+a);
    }
}
```



Non-Destructive Addition (Addition)

```java
public class Addition{
    Integer data;
    Addition(Integer d){
       data = d;
    }
    public Integer get(){
       return data;
    }
    public void set(Integer i){
        this.data = i;
    }
    public Integer add(Integer other){
       Integer result;
       result = this.data + (int) other;
       return result;
    }
    public Integer add(Addition other){
       Integer result;
       result = this.data + other.data;
       return result;
    }
    public String toString(){
      return data.toString();
    }

    public static void main(String[] args){
       Addition a = new Addition(3);
       Addition b = new Addition(4);
       System.out.println(a.add(b)+" "+a);
    }
}
```



Exercise 8:

Write Static add method and instance add method for Date Class

Assume 30 days a month, the answer may not make sense. We just want to perform addition. We don’t need to spend extra time to adjust it to get the result date correct.

Date8 Class

LECTURE 10

Pure methods

Pure methods

- This implementation of add does not modify either of the parameters. Instead, it creates and returns a new **Time** object. Alternatively, we could have written a method like this:

```
public void increment(double seconds) {
  this.second += seconds;
  while (this.second >= 60.0) {
    this.second -= 60.0;
    this.minute += 1;
  }
  while (this.minute >= 60) {
    this.minute -= 60;
    this.hour += 1;
 }
}
```

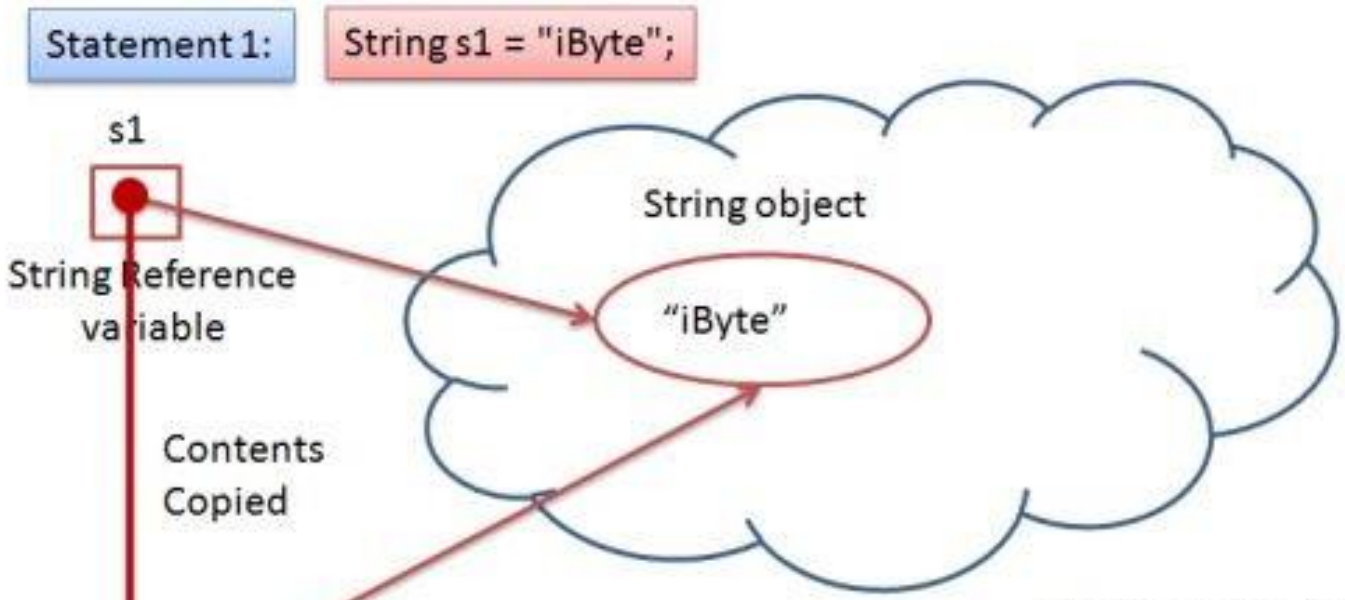
Pure methods

- The increment method modifies an existing **Time** object. It doesn't create a new one, and it doesn't return anything.
- In contrast, methods like add (in the previous section) are called **pure** because:
  - They don't modify the parameters.
  - They don't have any other "**side effects**", like printing.
  - The return value only depends on the parameters, not on any other data.
- Methods like **increment**, which breaks the first rule, are sometimes called **modifiers**. They are usually void methods, but sometimes they return a reference to the object they modify.

Pure methods

- Modifiers can be more efficient because they don't create new objects. But they can also be error-prone. When objects are aliased, the effects of modifiers can be confusing.
- If a class provides only getters and pure methods (no setters or modifiers), then the objects will be **immutable**. Working with immutable objects can be more difficult at first, but they can save you from long hours of debugging.

String Objects are Immutable



Statement 1 creates,
One reference variable, s1
one String object "iByte"

Statement 2 creates,
One reference variable, s2

String reference variables have bit-pattern (kind of pointer but NOT pointer) defining how to get to the String Object in the memory