Think Java

CHAPTER 9: IMMUTABLE OBJECTS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Java is an “object-oriented” language, which means that it uses objects to represent data and provide methods related to them. This way of **organizing programs** is a powerful design concept, and we will introduce it gradually throughout the remainder of the book.
- **An object is a collection of data that provides a set of methods.** For example, Scanner, which we saw in Section 3.2, is an object that provides methods for parsing input. System.out and System.in are also objects.

Objectives

- **Strings are objects**, too. They contain characters and provide methods for manipulating character data. Other data types, like Integer, contain numbers and provide methods for manipulating number data.
- We will explore some of those methods in this chapter.

Topics

- Primitive Data Versus Reference Data
- Stack Diagram
- String, Wrapper Classes, and BigInteger/BigDecimal Classes
- Command Line
- Software Development

LECTURE 1

Primitives vs objects

Primitive Data VS Reference Data

- Not everything in Java is an object: int, double, char, and boolean are examples of primitive types. When you declare a variable with a primitive type, Java reserves a small amount of memory to store its value. Figure 9.1 shows how the following values are stored memory.

```
int number = -2;

char symbol = '!';
```

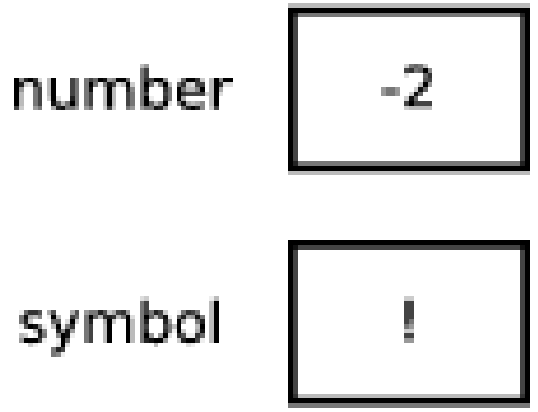


Figure 9.1: Memory diagram of two primitive variables.

Primitive Data VS Reference Data

- As we learned in Section 7.2, an array variable stores a reference to an array. That's because the array itself is too large to fit in the variable's memory. For example, char[] array = \{'c', 'a', 't'\}; contains three characters

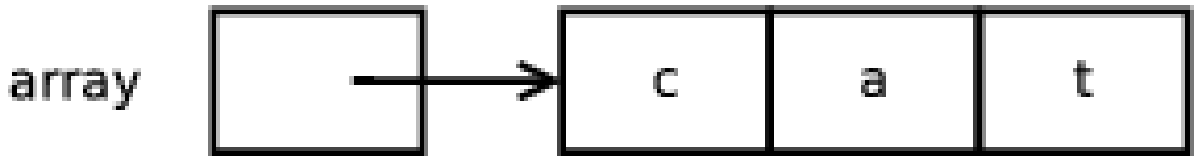


Figure 9.2: Memory diagram of an array of characters.

Memory Diagram

- When drawing memory diagrams, we use an arrow to represent the location of the array, as in Figure 9.2. The actual memory location (the value of the array variable) is an integer chosen by Java at run-time.
- Objects work in a similar way. When you declare an object variable, it will store a reference to an object. In contrast to arrays, which store multiple elements of the same data type, objects can be used to encapsulate any type of data.
- For example, a String object encapsulates a character array. Figure 9.3 illustrates how strings are stored in memory.
- Figure 9.3: Memory diagram of a String object.

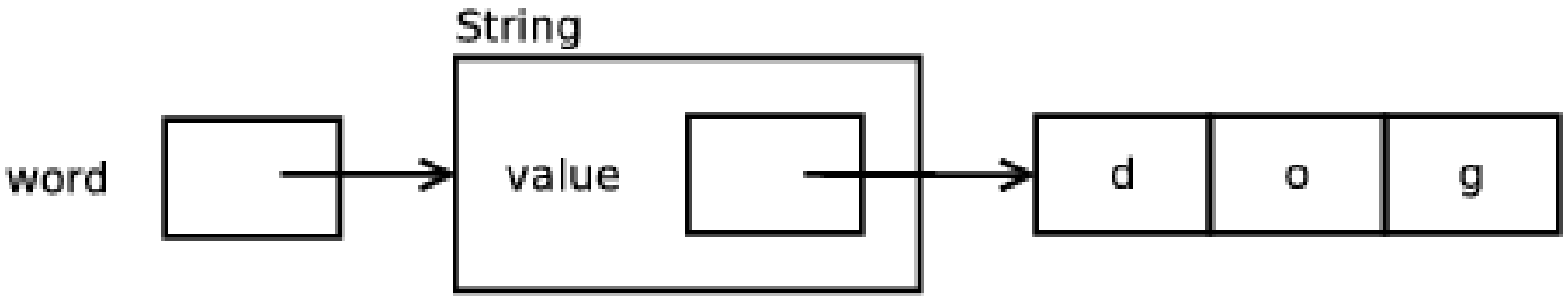
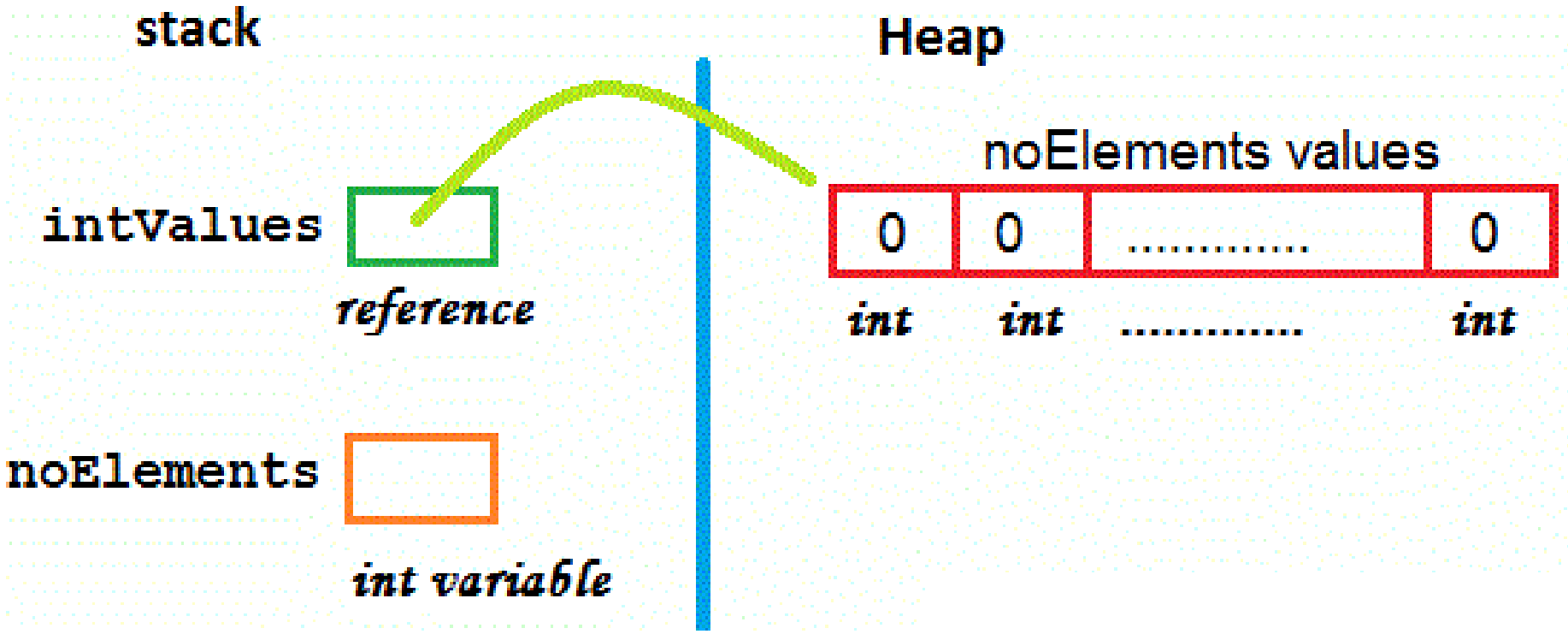
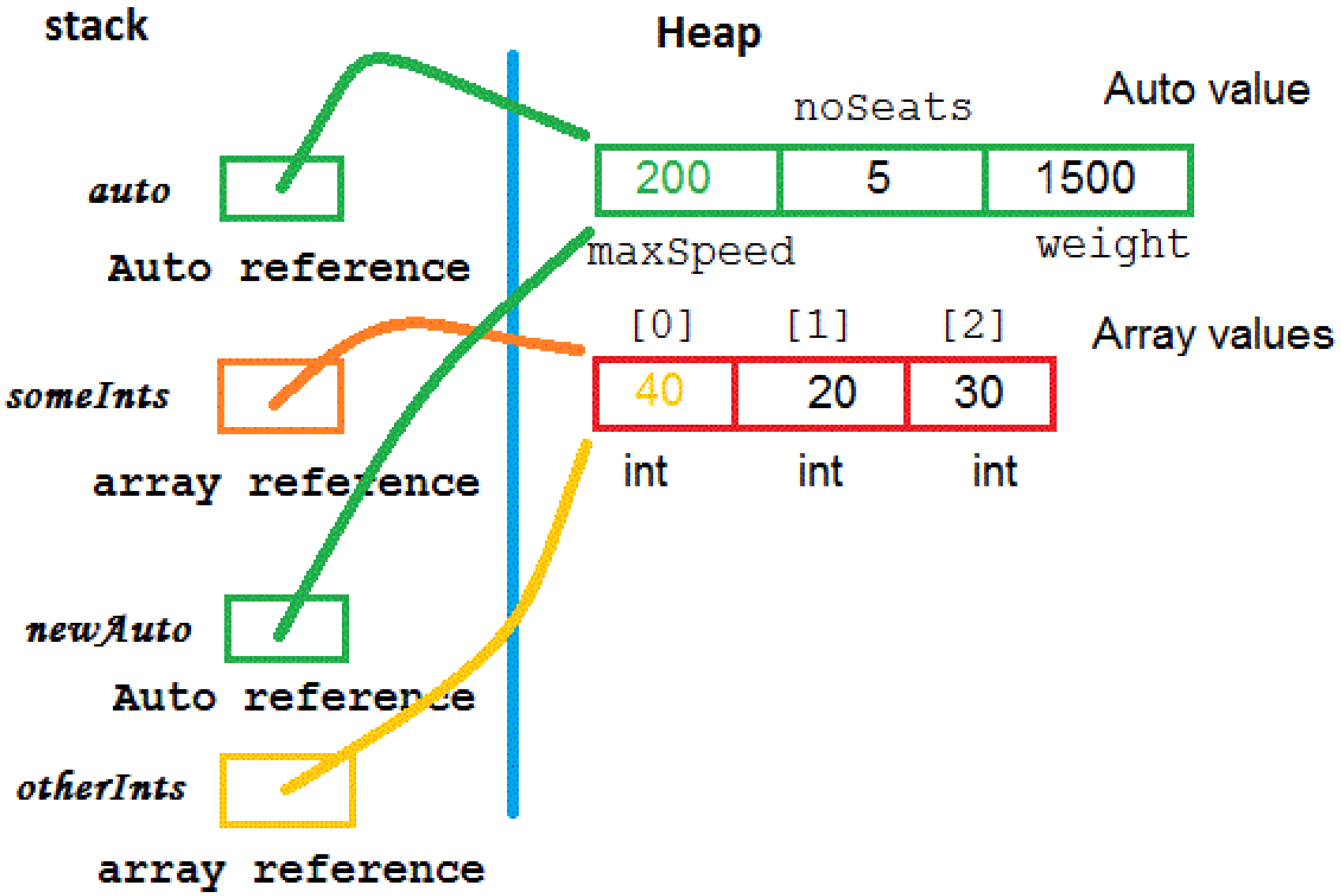


Figure 9.3: Memory diagram of a String object.


stack
Heap
noElements values
intValues
0
0
.....
0
reference
int
int
.....
int
noElements
int variable


stack
Heap
auto
Auto reference
someInts
array reference
newAuto
Auto reference
otherInts
array reference
noSeats
Auto value
200
5
1500
maxSpeed
weight
[0]
[1]
[2]
Array values
40
20
30
int
int
int

String as Reference Type

- Behind the scenes, the code String word = "dog"; creates an array of the characters 'd', 'o', and 'g', and stores the reference to that array in a String object. The variable word contains a reference to the String object.
- To test whether two integers (or other primitive types) are equal, you simply use the == operator. But as we learned in Section 6.8, you need to use the equals method to compare strings. The equals method traverses the arrays and tests whether they contain the same characters.

String as Reference Type

- On the other hand, two String objects with the same characters would not be considered equal in the == sense. The == operator, when applied to string variables, only tests whether they refer to the same object.
- In Java, the keyword null is a special value that means “no object”. You can initialize object and array variables this way:

```
String name = null;
int[] combo = null;
```

Null is an address with value of 0

- The value null is represented in memory diagrams by a small box with no arrow, as in Figure 9.4. In other words, the variables do not reference anything.

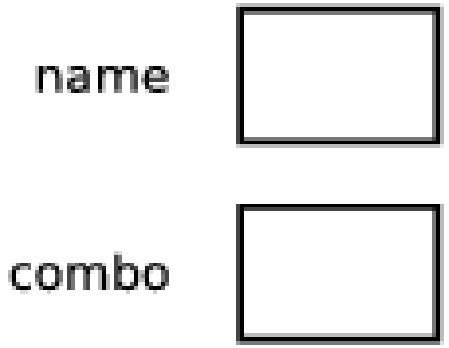


Figure 9.4: Memory diagram showing variables that are null.

NullPointerException

- If you try to use a variable that is null by invoking a method or accessing an element, Java throws a **NullPointerException**.

```
System.out.println(name.length());   //
NullPointerException
System.out.println(combo[0]);         //
NullPointerException
```

- On the other hand, it is perfectly fine to pass a null reference as an argument to a method, or to receive one as a return value. In these situations, null is often used to represent a special condition or indicate an error

LECTURE 2

Strings are immutable

String Class

- If the Java library didn’t have a String class, we would have to use character arrays to store and manipulate text. Operations like concatenation (+), indexOf, and substring would be difficult and inconvenient. Fortunately, Java does have a String class that provides these and other methods.
- For example, the methods **toLowerCase** and **toUpperCase** convert uppercase letters to lowercase, and vice versa. These methods are often a source of confusion, because it sounds like they modify strings. But neither these methods nor any others can change a string, because strings are **immutable**.

String Functions

- When you invoke **toUpperCase** on a string, you get another String object as a result. For example:

```
String name = "Alan Turing";
String upperName = name.toUpperCase();
```

- After these statements run, **upperName** refers to the string "ALAN TURING". But name still refers to "Alan Turing".

Redirection of String Reference

- A common mistake is to assume that toUpperCase somehow affects the original string:

```
String name = "Alan Turing";
name.toUpperCase();//ignores the return value
System.out.println(name);
```

Redirection of String Reference

- The previous code displays "Alan Turing", because the value of name (i.e., the reference to the original String object) never changes. If you want to change name to be uppercase, then you need to assign the return value to it:

```
String name = "Alan Turing";
name = name.toUpperCase();// references the
                          // new string
System.out.println(name);
```

Replace Method

- A similar method is replace, which finds and replaces instances of one string within another. This example replaces "Computer Science" with "CS":

```
String text = "Computer Science is fun!";

text = text.replace("Computer Science", "CS");
```

- As with toUpperCase, assigning the return value (to text) is important. If you don’t assign the return value, invoking text.replace has no effect.

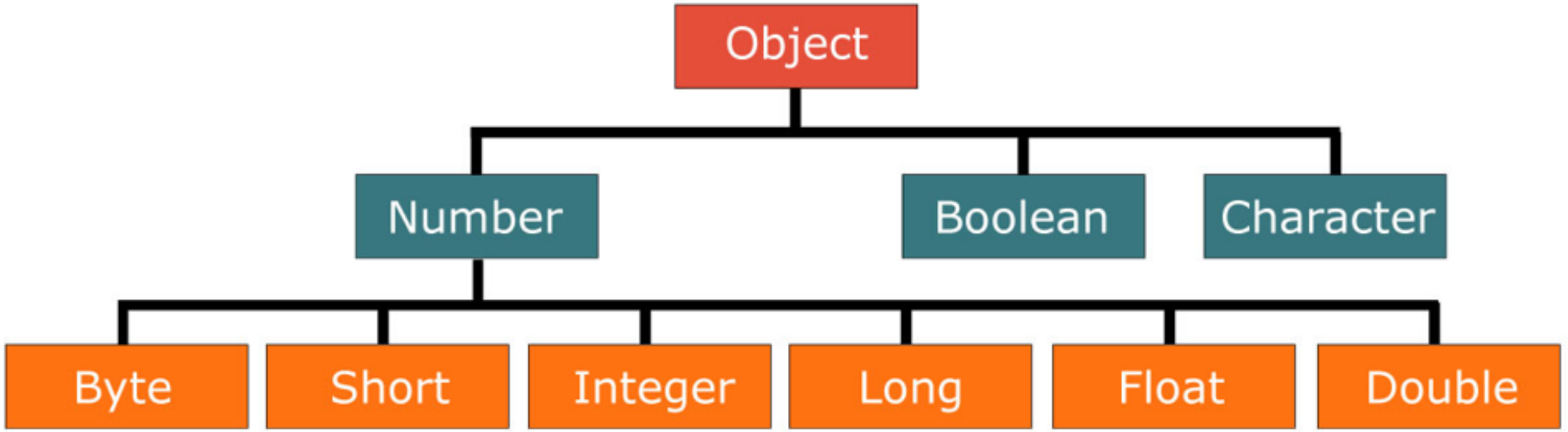
Operations on Strings are Non-Destructive

Strings are immutable by design, because it simplifies passing them between methods as parameters and return values. And since the contents of a string can never change, two variables can reference the same string without one accidentally corrupting the other.

LECTURE 3

Wrapper classes

Wrapper Class Hierarchy



Wrapper Classes

- Primitive values (like ints, doubles, and chars) cannot be null, and they do not provide methods. For example, you can't invoke equals on an int:

```
int i = 5;
System.out.println(i.equals(5));// compiler error
```

- But for each primitive type, there is a corresponding wrapper class in the Java library. The wrapper class for int is named Integer, with a capital I.

```
Integer i = new Integer(5);
System.out.println(i.equals(5));// displays true
```

Wrapper Classes

- Other wrapper classes include **Boolean**, **Character, Double**, and **Long**. They are in the java.lang package, so you can use them without importing them.
- Like strings, objects from wrapper classes are immutable. And you need to use the equals method to compare them.

```
Integer x = new Integer(123);
Integer y = new Integer(123);
if (x == y) {                              // false
    System.out.println("x and y are the same object");
}
if (x.equals(y)) {                         // true
    System.out.println("x and y have the same value");
}
```

Constants in Wrapper Class

- Because x and y refer to different Integer objects, the code only displays “x and y have the same value”.
- Each wrapper class defines the constants MIN_VALUE and MAX_VALUE. For example, Integer.MIN_VALUE is -2147483648, and Integer.MAX_VALUE is 2147483647. Because these constants are available in wrapper classes, you don’t have to remember them, and you don’t have to write them yourself.
- Wrapper classes also provide methods for converting strings to and from primitive types. For example, Integer.parseInt converts a string to (you guessed it) an integer. In this context, parse means “read and translate”.

Wrapper Data Types and Conversion

- String str = "12345";

```
int num = Integer.parseInt(str);
```

- The other wrapper classes provide similar methods, like **Double.parseDouble** and **Boolean.parseBoolean**. They also each provide toString, which returns a string representation of a value:

```
int num = 12345;

String str = Integer.toString(num);
```

- The result is the string "12345", which as you now understand, is stored internally in a character array \{'1', '2', '3', '4', '5'\}.

Wrapper Data Types and Conversion

- It’s always possible to convert a primitive value to a string, but not the other way around. The following code throws a **NumberFormatException**.

```
String str = "five";

int num = Integer.parseInt(str);  // NumberFormatException
```

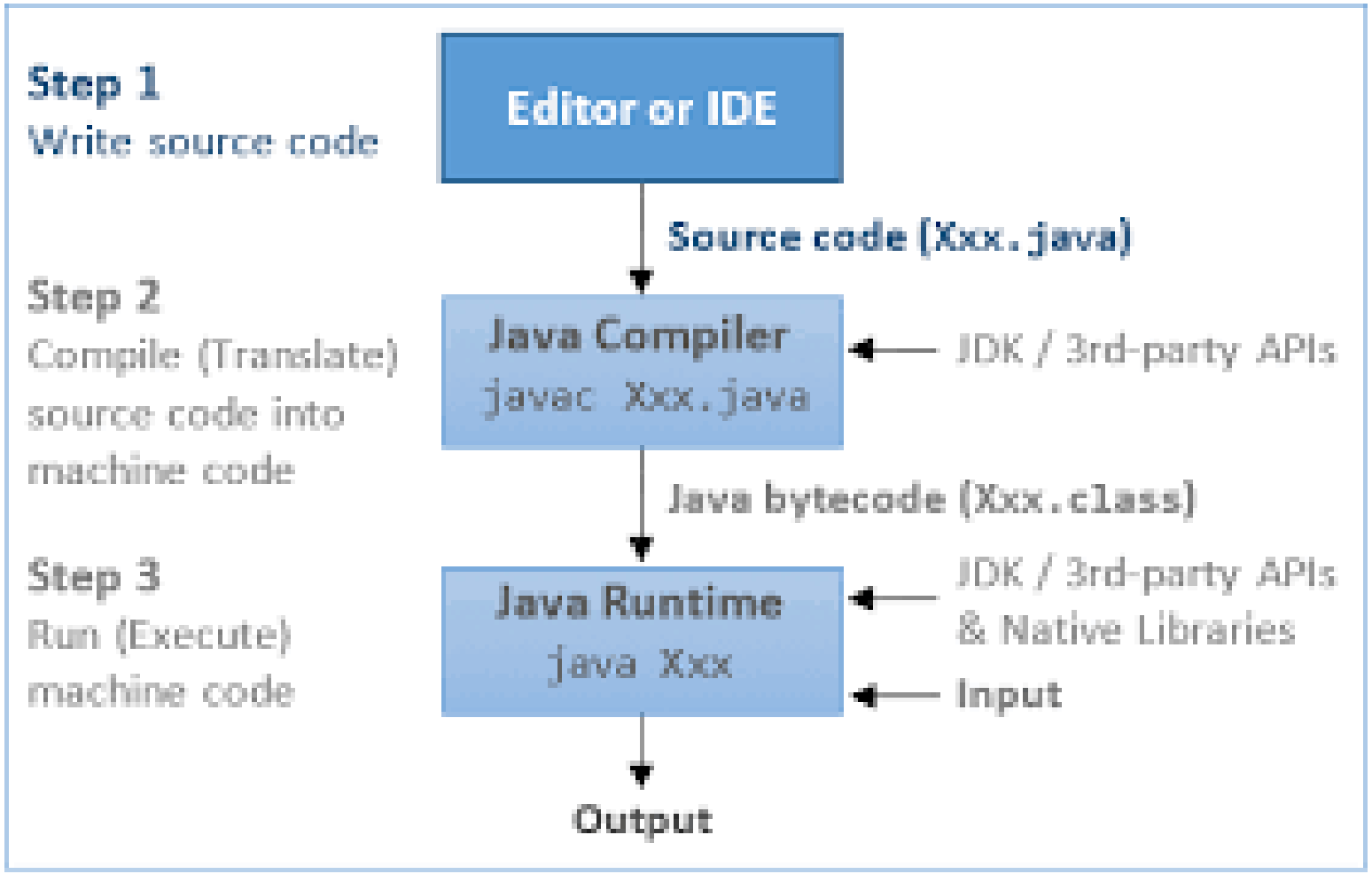
In-Class Demo Program

- String Operation “ab” to “bbaa” (reverse and double)

MAX.JAVA

LECTURE 4

Command-line arguments


Step 1
Write source code
Editor or IDE
Source code (Xxx.java)
Step 2
Compile (Translate)
source code into
machine code
Java Compiler
javac Xxx.java
JDK / 3rd-party APIs
Java bytecode (Xxx.class)
Step 3
Run (Execute)
machine code
Java Runtime
java Xxx
JDK / 3rd-party APIs
& Native Libraries
Input
Output

Command-line Execution

- Let's write a program to find the maximum value in a sequence of numbers. Rather than read the numbers from System.in using a Scanner, we'll pass them as command-line arguments. Here is a starting point:

```
public class Max {
    public static void main(String[] args) {
        System.out.println(Arrays.toString(args));
    }
}
```

- You can run this program from the command line by typing:

```
java Max
```

Command-line Execution

- The output indicates that args is an empty array; that is, it has no elements:

```
[]
```

- If you provide additional values on the command line, they are passed as arguments to main. For example, if you run the program like this:

```
java Max 10 -3 55 0 14
```

Command-line Execution

- The output is:

```
[10, -3, 55, 0, 14]
```

- It's not clear from the output, but the elements of args are strings.
- So args is the array \{"10", "-3", "55", "0", "14"\}. To find the maximum number, we have to convert the arguments to integers.

Finding Maximum Value

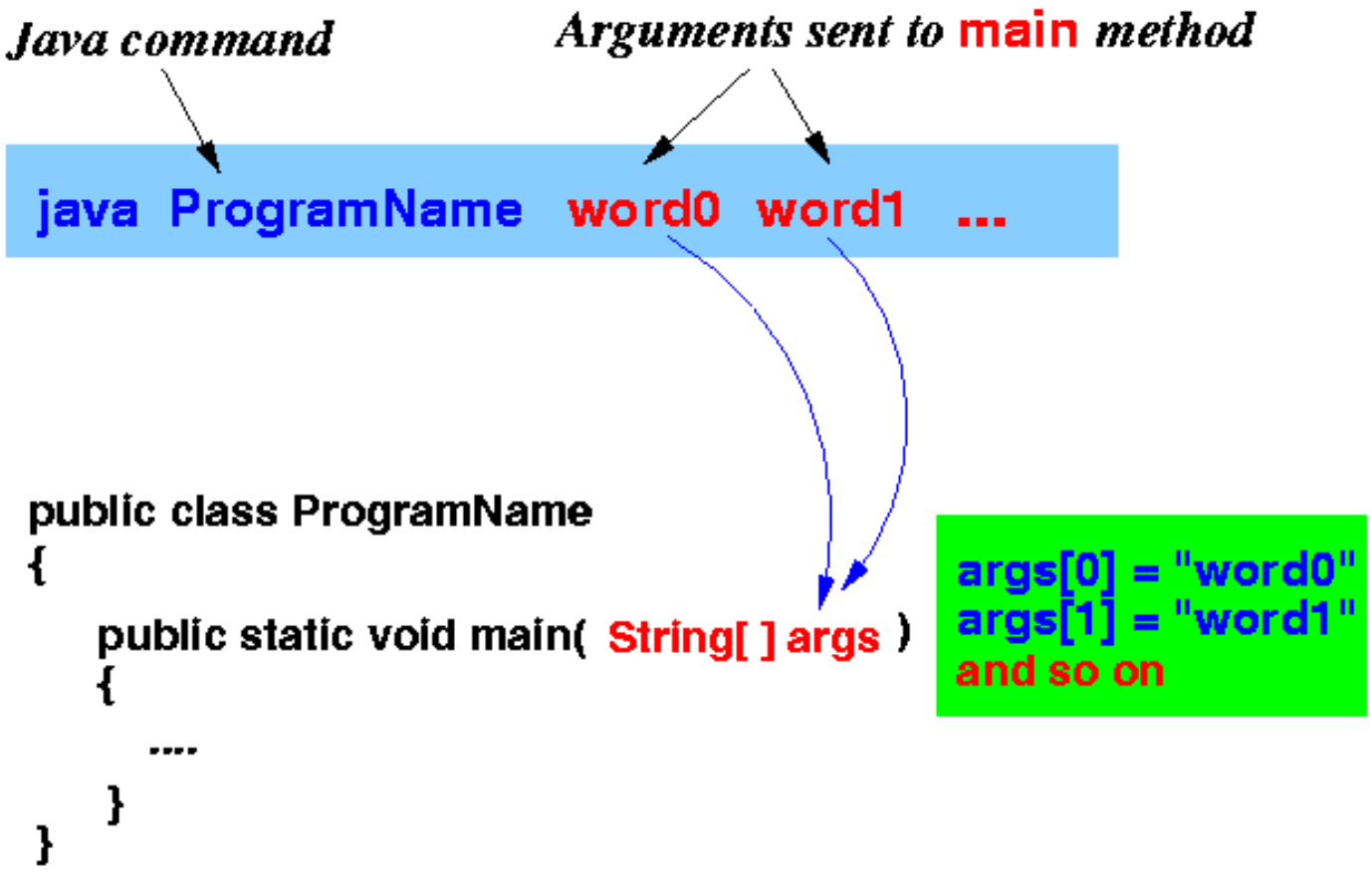
- The following code uses an enhanced for loop to parse the arguments (using the Integer wrapper class) and find the largest value:

```
int max = Integer.MIN_VALUE;
for (String arg : args) {
    int value = Integer.parseInt(arg);
    if (value > max) {
        max = value;
    }
}
System.out.println("The max is " + max);
```

Command-line Execution

- It’s customary for programs that require command-line arguments to display a “usage” message when there are no arguments given. For example, if you run **javac** or **java** from the command line without any arguments, you will get a very long message.

Java Command Line Arguments



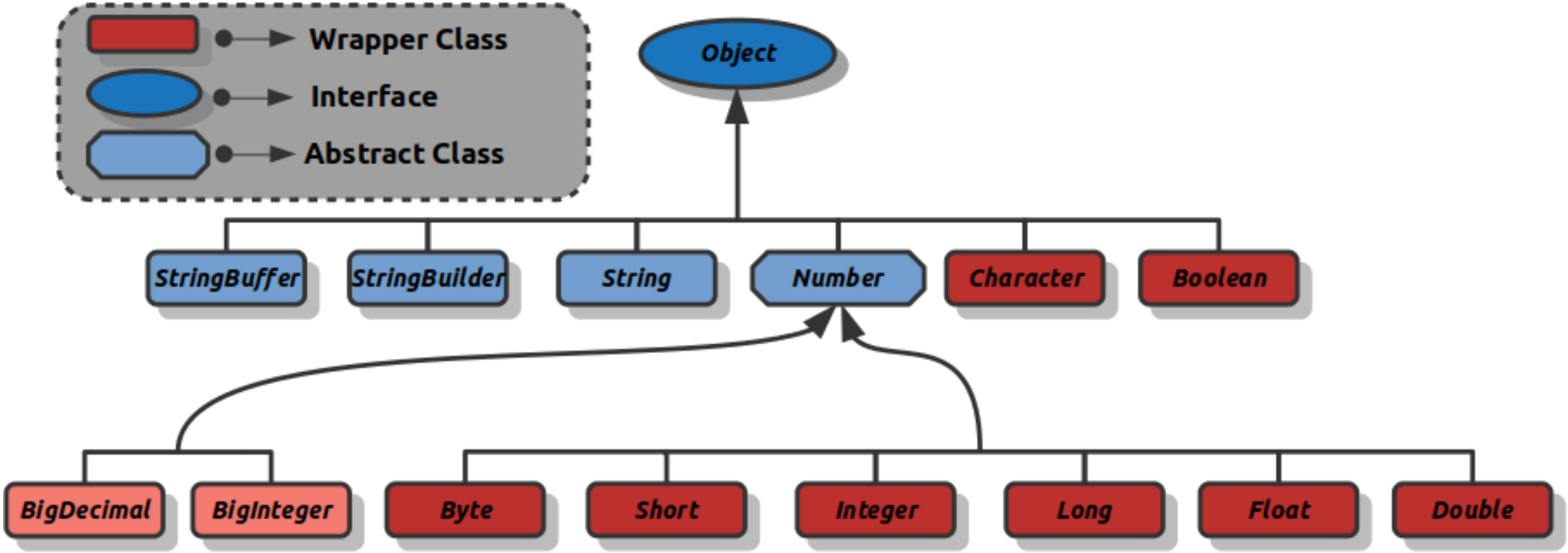
In-Class Demo Program

- Get input for Hour, Minutes and Second in a Time Card.
- Provide addition, Subtraction methods
- toString(), equals()
- Command Line Compilation Flow

TIME.JAVA

LECTURE 5

BigInteger arithmetic


Wrapper Class
Interface
Abstract Class
Object
StringBuffer
StringBuilder
String
Number
Character
Boolean
BigDecimal
BigInteger
Byte
Short
Integer
Long
Float
Double

BigInteger

- It might not be clear at this point why you would ever need an integer object when you can just use an int or long. One advantage is the variety of methods that Integer and Long provide. But there is another reason: when you need very large integers that exceed **Long.MAX_VALUE**.
- **BigInteger** is a Java class that can represent arbitrarily large integers. There is no upper bound except the limitations of memory size and processing speed. Take a minute to read the documentation, which you can find by doing a web search for “Java BigInteger”.

BigInteger

- To use BigIntegers, you have to import **java.math.BigInteger** at the beginning of your program. There are several ways to create a BigInteger, but the simplest uses **valueOf**. The following code converts a long to a BigInteger:

```
long x = 17;
BigInteger big = BigInteger.valueOf(x);
```

- You can also create BigIntegers from strings. For example, here is a 20-digit integer that is too big to store using a long.

```
String s = "12345678901234567890";
BigInteger bigger = new BigInteger(s);
```

BigInteger

- Notice the difference in the previous two examples: you use valueOf to convert integers, and new BigInteger to convert strings.
- Since BigIntegers are not primitive types, the usual math operators don't work. Instead, we have to use methods like add. To add two BigIntegers, we invoke add on one and pass the other as an argument.

```
BigInteger a = BigInteger.valueOf(17);
BigInteger b =
BigInteger.valueOf(1700000000);
BigInteger c = a.add(b);
```

BigInteger

- Like strings, BigInteger objects are **immutable**. Methods like add, multiply, and pow all return new BigIntegers, rather than modify an existing one.
- Internally, a BigInteger encapsulates an array of ints, similar to the way a string encapsulates an array of chars. Each int in the array stores a portion of the BigInteger. The methods of BigInteger traverse this array to perform addition, multiplication, etc.
- For very long floating-point values, take a look at **java.math.BigDecimal**. Interestingly, BigDecimal objects represent floating-point numbers internally by encapsulating a BigInteger!

In-Class Demo Program

- Create a PI class to create an BigDecimal object.
- Demo Simple Operations

PI.JAVA

Project: PI.java
(Student in-Class Project)

For estimating π

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

LECTURE 6

Program development

Encapsulation and Generalization

- This chapter introduces two main concepts:
  1. objects **encapsulate** other types of data, and
  2. they can be designed to be **immutable**.
- Applying these concepts helps us to manage the complexity of programs as they become large.
- Unfortunately, computer science has a lot of overloaded terms. Another use of the term “encapsulation” applies to methods. In this section, we present a design process called “encapsulation and generalization”.

Encapsulation and Generalization

- One challenge of programming, especially for beginners, is figuring out how to divide up a program into methods. The process of **encapsulation** and **generalization** allows you to design as you go along. The steps are:
    1. Write a few lines of code in main or another method, and test them.
    2. When they are working, wrap them in a new method, and test again.
    3. If it's appropriate, replace literal values with variables and parameters.

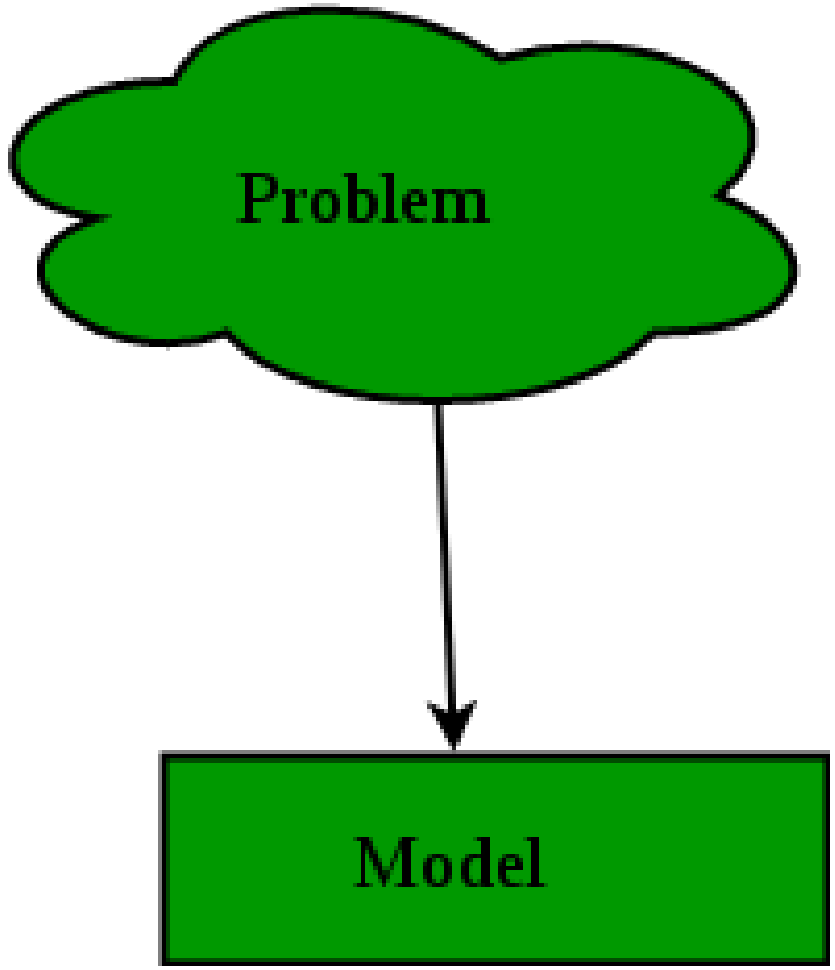
- Encapsulation and generalization is similar to "incremental development" (see Section 4.8), in the sense that you write a little code, test it, and repeat. But you don't need to begin with an exact method definition and stub.

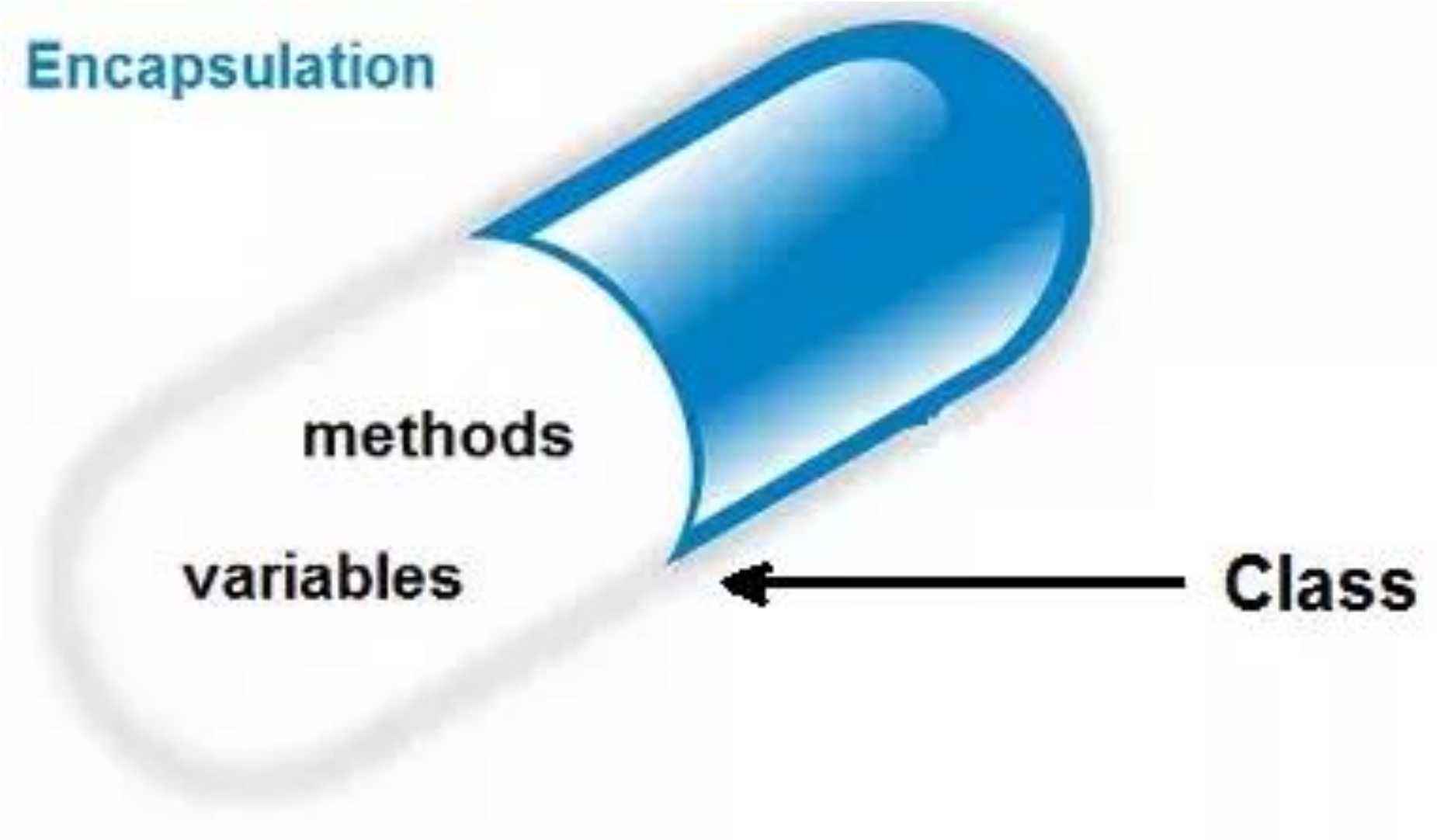
Encapsulation and Generalization

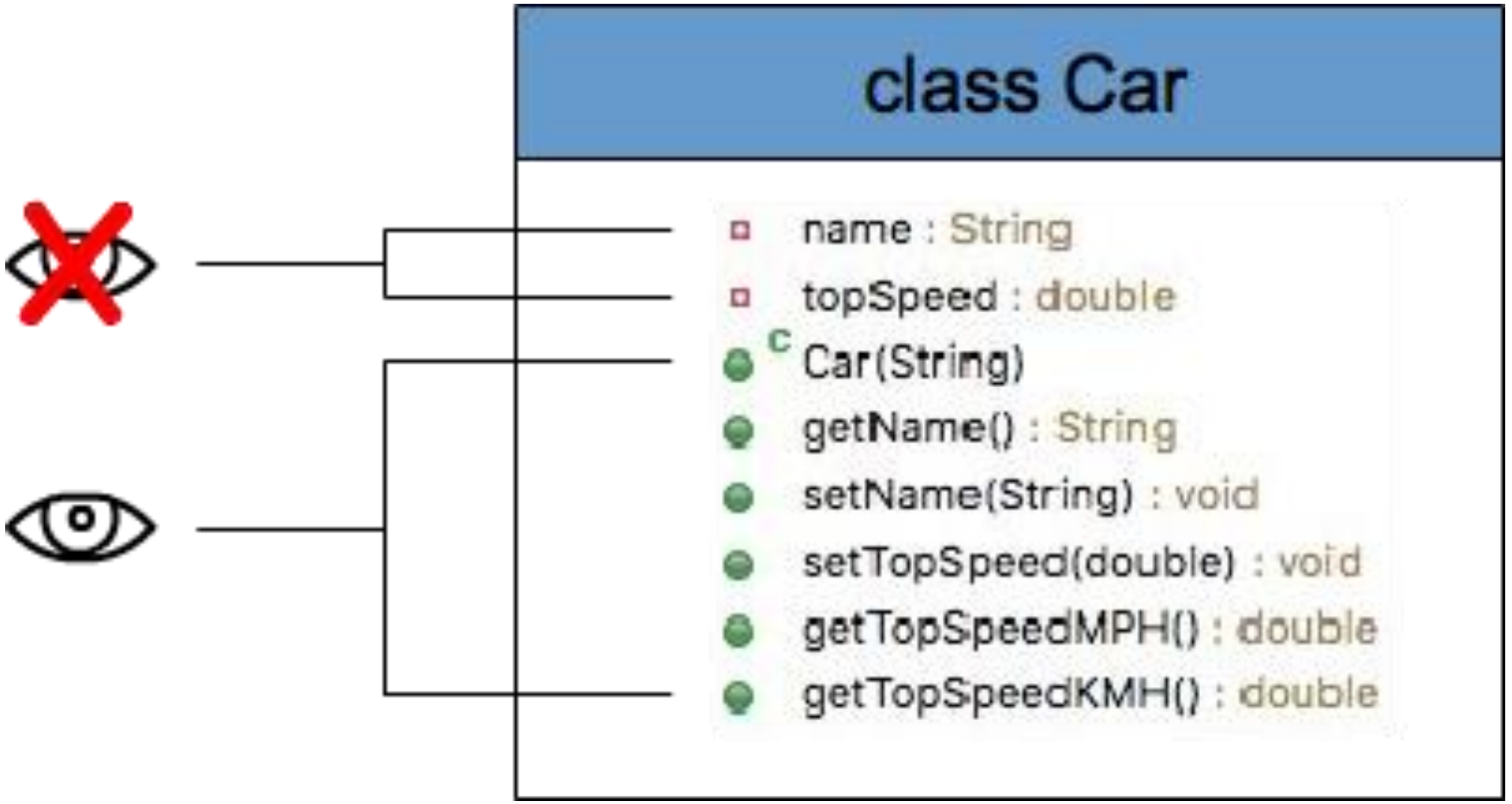
- To demonstrate this process, we'll develop methods that display **multiplication tables**. Here is a loop that displays the multiples of two, all on one line:

```
int i = 1;
while (i <= 6) {
    System.out.printf("%4d", 2 * i);
    i = i + 1;
}
System.out.println();
```

- The first line initializes a variable named i, which is going to act as the loop variable. As the loop executes, the value of i increases from 1 to 6; when i is 7, the loop terminates.


Problem
Model


Encapsulation
methods
variables
Class


class Car
name : String
topSpeed : double
Car(String)
getName() : String
setName(String) : void
setTopSpeed(double) : void
getTopSpeedMPH() : double
getTopSpeedKMH() : double

Multiplication Table

- Each time through the loop, we display the value **2 * i** padded with spaces so it's four characters wide. Since we use **System.out.printf**, the output appears on a single line.
- After the loop, we call **println** to print a newline and complete the line. Remember that in some environments, none of the output is displayed until the line is complete.
- The output of the code so far is:

```
2   4   6   8   10   12
```

Encapsulation

- The next step is to **encapsulate** or wrap this code in a method. Here's what it looks like:

```
public static void printRow() {
   int i = 1;
   while (i <= 6) {
      System.out.printf("%4d", 2 * i);
      i = i + 1;
   }
   System.out.println();
}
```

Generalization

- Next, we generalize the method by replacing the constant value, 2, with a parameter, n. This step is called “generalization” because it makes the method more general (less specific).

```
public static void printRow(int n) {
    int i = 1;
    while (i <= 6) {
        System.out.printf("%4d", n * i);//generalized n
        i = i + 1;
    }
    System.out.println();
}
```

- Invoking this method with the argument 2 yields the same output as before. With the argument 3, the output is:

```
   3   6   9  12  15  18
```

Multiplication Table

- By now you can probably guess how we are going to display a multiplication table: we'll invoke printRow repeatedly with different arguments. In fact, we'll use another loop to iterate through the rows.

```
int i = 1;
while (i <= 6) {
    printRow(i);
    i = i + 1;
}
```

Multiplication Table

- And the output looks like this:

```
1   2   3   4   5   6
2   4   6   8  10  12
3   6   9  12  15  18
4   8  12  16  20  24
5  10  15  20  25  30
6  12  18  24  30  36
```

In-Class Demo Program

- 9x9 Multiplication Table

MTABLE.JAVA

LECTURE 7

More generalization

Nested Loop

- The previous result is similar to the “nested loops” approach in Section 6.4. However, the inner loop is now encapsulated in the **printRow** method. We can encapsulate the outer loop in a method too:

```
public static void printTable() {
    int i = 1;
    while (i <= 6) {
        printRow(i);
        i = i + 1;
    }
}
```

Abstraction with Parameter

Generalization - Abstraction

- The initial version of printTable always displays six rows. We can generalize it by replacing the literal 6 with a parameter:

```
public static void printTable(int rows) {
    int i = 1;
    while (i <= rows) {   // generalized rows
        printRow(i);
        i = i + 1;
    }
}
```

Output for printTable

- Here is the output of printTable(7):

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36
7	14	21	28	35	42

Programmable Row/Col Numbers

- That's better, but it still has a problem: it always displays the same number of columns. We can generalize more by adding a parameter to **printRow**:

```
public static void printRow(int n, int cols) {
    int i = 1;
    while (i <= cols) {  // generalized cols
        System.out.printf("%4d", n * i);
        i = i + 1;
    }
    System.out.println();
}
```

printRow

- Now printRow takes two parameters: n is the value whose multiples should be displayed, and cols is the number of columns. Since we added a parameter to printRow, we also have to change the line in printTable where it is invoked:

```
public static void printTable(int rows) {
    int i = 1;
    while (i <= rows) {
        printRow(i, rows);  // added rows argument
        i = i + 1;
    }
}
```

printRow

- When this line executes, it evaluates rows and passes the value, which is 7 in this example, as an argument. In **printRow**, this value is assigned to cols. As a result, the number of columns equals the number of rows, so we get a square 7x7 table (instead of the previous 7x6 table):
- When you generalize a method appropriately, you often find that it has capabilities you did not plan. For example, you might notice that the multiplication table is symmetric. Since **ab = ba**, all the entries in the table appear twice. You could save ink by printing half of the table, and you would only have to change one line of **printTable**:

```
printRow(i, i);  // using i for both n and cols
```

Triangular Multiplication

- In English, the length of each row is the same as its row number. The result is a triangular multiplication table.

```
1
2  4
3  6  9
4  8  12  16
5  10  15  20  25
6  12  18  24  30  36
7  14  21  28  35  42  49
```

- Generalization makes code more versatile, more likely to be reused, and sometimes easier to write. In this example, we started with a simple idea and ended with two general-purpose methods.

In-Class Demo Program

- 2D Index Space

TWOD.JAVA

Homework

Homework

- Textbook Exercise: Chapter 9
- Project 9