Chapter 10 Thinking in Objects

Updated resource site (git):
https://github.com/huzhengatUCSD/Java_Course.git

(ftp):
buptjava:buptjavastudent@ftp.redwing.buptnet.edu.cn:8021
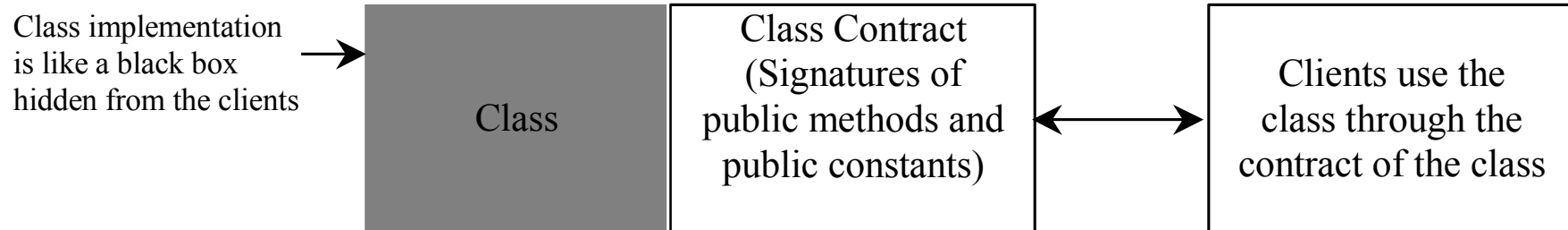
Motivations

You see the advantages of object-oriented programming from the preceding chapter. This chapter will demonstrate how to solve problems using the object-oriented paradigm.

Objectives

- To apply class abstraction to develop software (§10.2).
- To explore the differences between the procedural paradigm and object -oriented paradigm (§10.3).
- To discover the relationships between classes (§10.4).
- To design programs using the object-oriented paradigm (§§10.5–10.6).
- To create objects for primitive values using the wrapper classes (**Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, and **Boolean**) (§10.7).
- To simplify programming using automatic conversion between primitive types and wrapper class types (§10.8).
- To use the **BigInteger** and **BigDecimal** classes for computing very large numbers with arbitrary precisions (§10.9).
- To use the **String** class to process immutable strings (§10.10).
- To use the **StringBuilder** and **StringBuffer** classes to process mutable strings (§10.11).

Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.

Class implementation is like a black box hidden from the clients → | Class | Class Contract (Signatures of public methods and public constants) ↔ Clients use the class through the contract of the class

Designing the Loan Class

| Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1)
-loanAmount: double	The loan amount (default: 1000).
-loanDate: Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years to this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

Loan   TestLoanClass

Object-Oriented Thinking

Chapters 1-8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. The studies of these techniques lay a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software.

From the improvements, you will gain the insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.

The BMI Class
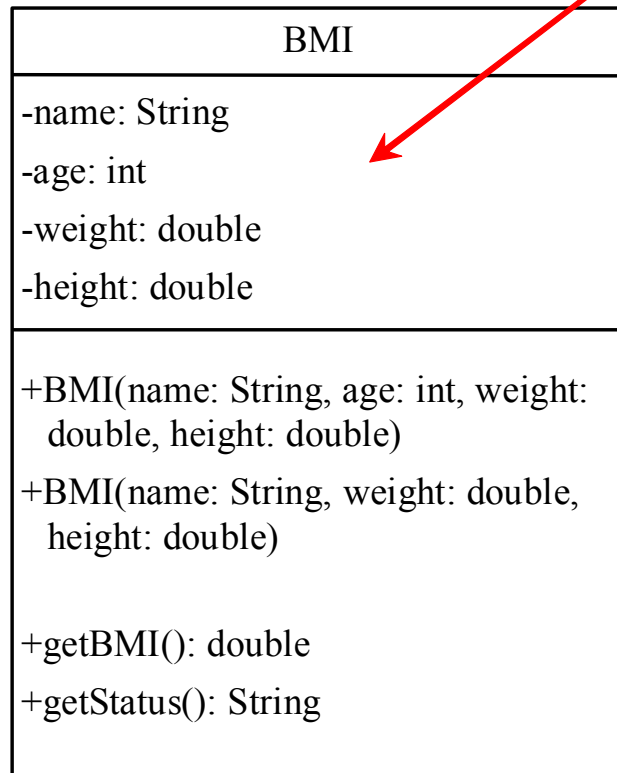
The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

BMI
-name: String
-age: int
-weight: double
-height: double
+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+getBMI(): double
+getStatus(): String

The name of the person.

The age of the person.

The weight of the person in pounds.

The height of the person in inches.

Creates a BMI object with the specified name, age, weight, and height.

Creates a BMI object with the specified name, weight, height, and a default age 20.

Returns the BMI

Returns the BMI status (e.g., normal, overweight, etc.)

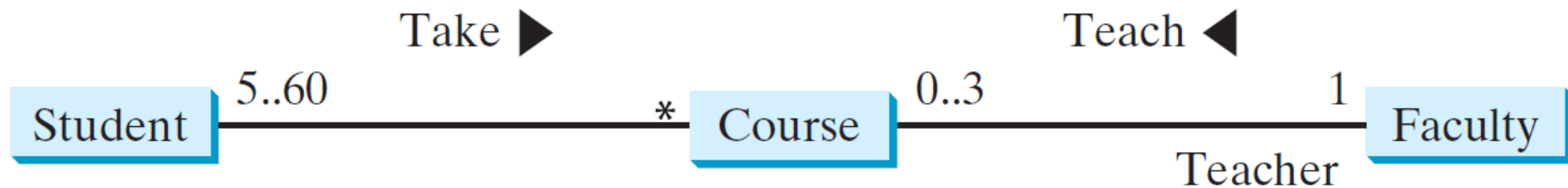BMI   UseBMIClass

Relationship between Classes

✦ Association

   – Aggregation

      ◆ Composition

Association

Relationship between two objects/classes

Describe the activity between two objects/classes

Take ▶   Teach ◀

```
         5..60                      0..3                    1
Student ─────────────── * Course ──────────────── Faculty
                                              Teacher
```

```
public class Student {
  private Course[]
    courseList;
  public void addCourse(
    Course s) {…}
}
```

```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;
  ...
}
```

```
public class Faculty {
  private Course[]
    courseList;
  public void addCourse(
    Course s) {…}
}
```

Aggregation

Aggregation is actually a special case of the Association relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.

Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class.

```
public class Name {

  ...

}
```

Aggregated class

```
public class Student {

  private Name name;

  private Address address;


  ...

}
```

Aggregating class

```
public class Address {

  ...

}
```
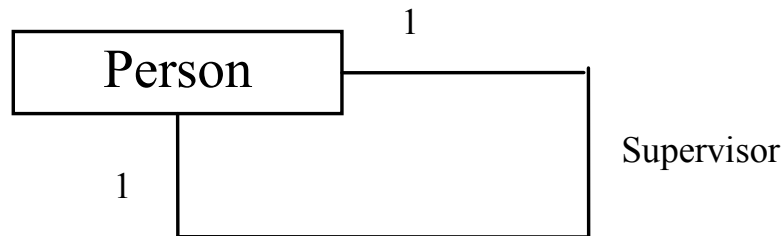
Aggregated class

Aggregation vs. Composition

组合

聚集

名字 **1** **1** 学生 **1..3** **1** 地址
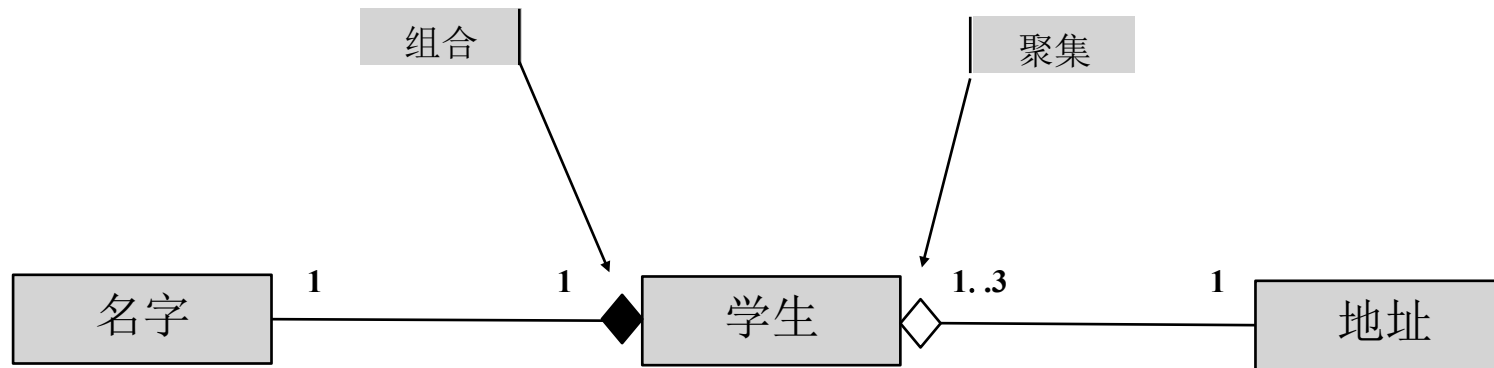
Aggregation Between Same Class

Aggregation may exist between objects of the same class.
For example, a person may have a supervisor.

```
          1
┌──────────────┐
│    Person    │───────────────┐
└──────────────┘               │ Supervisor
    1     │                     │
          └─────────────────────┘
```

```
public class Person {
    // The type for the data is the class itself
    private Person supervisor;
    ...
}
```

Aggregation Between Same Class

What happens if a person has several supervisors?

Person

1

Supervisor

m
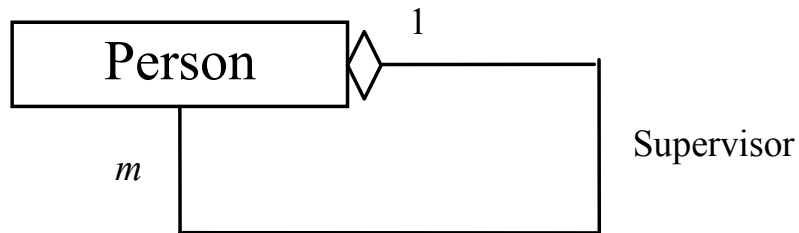
```
public class Person {
   ...
   private Person[] supervisors;
}
```

Example: The Course Class

Course
-courseName: String
-students: String[]
-numberOfStudents: int
+Course(courseName: String)
+getCourseName(): String
+addStudent(student: String): void
+dropStudent(student: String): void
+getStudents(): String[]
+getNumberOfStudents(): int

The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.

Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.
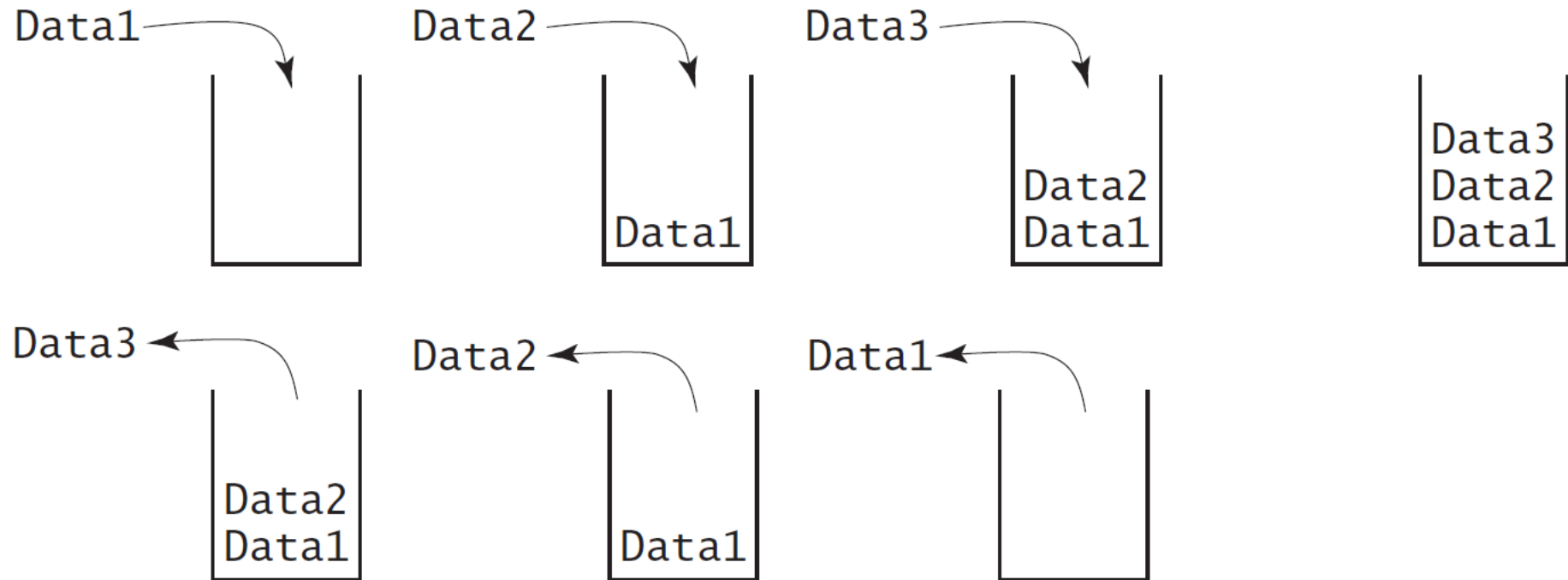
Course   TestCourse

Example: The StackOfIntegers Class

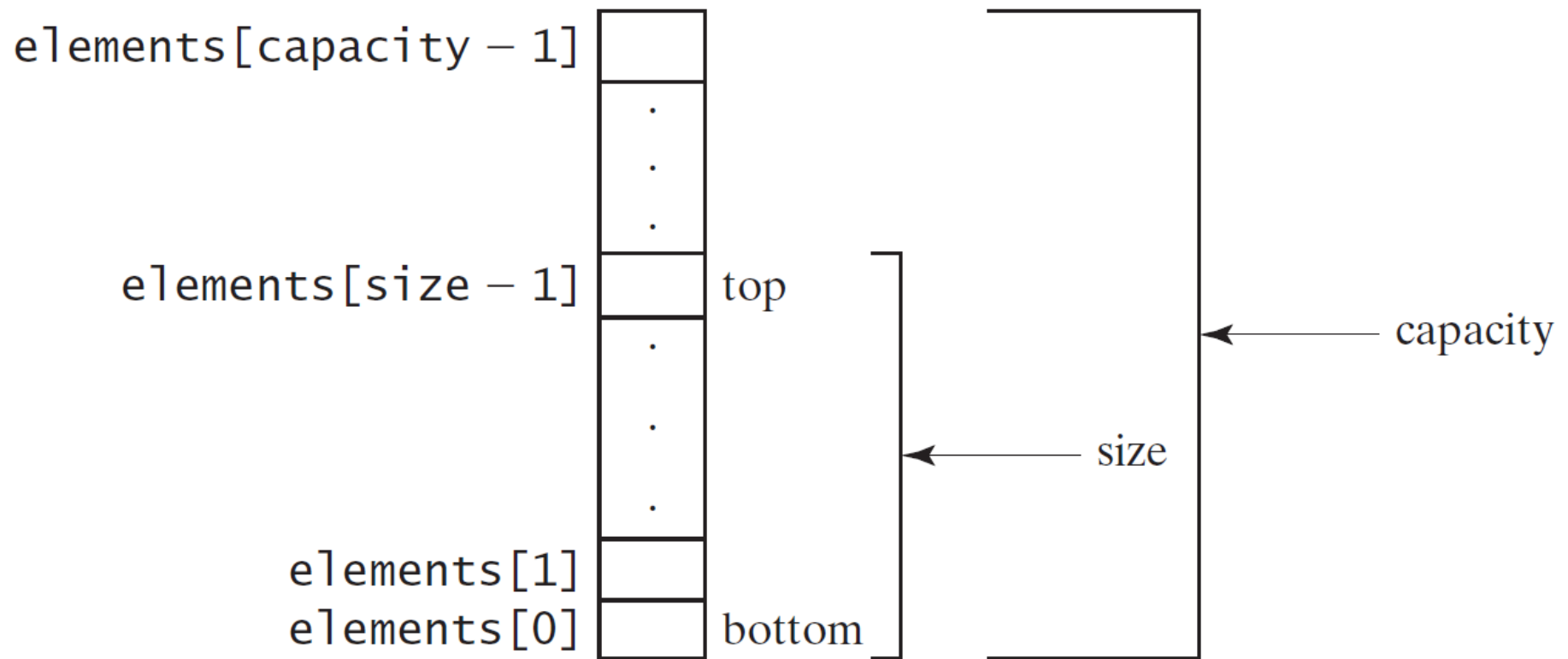| StackOfIntegers	
-elements: int[]	An array to store integers in the stack.
-size: int	The number of integers in the stack.
+StackOfIntegers()	Constructs an empty stack with a default capacity of 16.
+StackOfIntegers(capacity: int)	Constructs an empty stack with a specified capacity.
+empty(): boolean	Returns true if the stack is empty.
+peek(): int	Returns the integer at the top of the stack without removing it from the stack.
+push(value: int): int	Stores an integer into the top of the stack.
+pop(): int	Removes the integer at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

TestStackOfIntegers

Designing the StackOfIntegers Class

Implementing StackOfIntegers Class

elements[capacity − 1]

elements[size − 1]   top

elements[1]

elements[0]   bottom

capacity

size

StackOfIntegers

Wrapper Classes

- Boolean

- Character

- Short

- Byte

- Integer

- Long

- Float

- Double

NOTE: (1) The wrapper classes do not have no-arg constructors. (2) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.

The `Integer` and `Double` Classes

java.lang.Integer
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue():double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int

java.lang.Double
-value: double
+MAX_VALUE: double
+MIN_VALUE: double
+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue():double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+parseDouble(s: String): double

The `Integer` Class
and the `Double` Class

❑ Constructors

❑ Class Constants `MAX_VALUE, MIN_VALUE`

❑ Conversion Methods

Numeric Wrapper Class Constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

public Integer(int value)

public Integer(String s)

public Double(double value)

public Double(String s)

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants
MAX_VALUE and MIN_VALUE. MAX_VALUE
represents the maximum value of the corresponding
primitive data type. For Byte, Short, Integer, and Long,
MIN_VALUE represents the minimum byte, short, int,
and long values. For Float and Double, MIN_VALUE
represents the minimum *positive* float and double
values. The following statements display the maximum
integer (2,147,483,647), the minimum positive float
(1.4E-45), and the maximum double floating-point
number (1.79769313486231570e+308d).

Conversion Methods

Each numeric wrapper class implements the abstract methods <u>doubleValue</u>, <u>floatValue</u>, <u>intValue</u>, <u>longValue</u>, and <u>shortValue</u>, which are defined in the <u>Number</u> class. These methods "convert" objects into primitive type values.

The Static <u>valueOf</u> Methods

The numeric wrapper classes have a useful class method, valueOf(String s). This method creates a new object initialized to the value represented by the specified string. For example:

```
Double doubleObject = Double.valueOf("12.4");

Integer integerObject = Integer.valueOf("12");
```
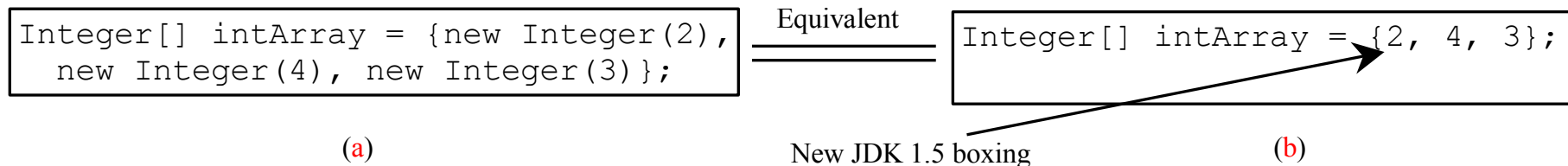
The Methods for Parsing Strings into Numbers

Use the parseInt method in the Integer class to parse a numeric string into an int value and

the parseDouble method in the Double class to parse a numeric string into a double value.

Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):

```
Integer[] intArray = {new Integer(2),
   new Integer(4), new Integer(3)};
```

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

(a)

New JDK 1.5 boxing

(b)

Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);

Unboxing

BigInteger and BigDecimal

To compute with very large integers or high precision floating-point values, use the BigInteger and BigDecimal classes in the java.math package.

Both are *immutable*.

Both extend the Number class and implement the Comparable interface.

BigInteger and BigDecimal

BigInteger a = **new** BigInteger("9223372036854775807");

BigInteger b = **new** BigInteger("2");

BigInteger c = a.multiply(b); // 9223372036854775807 * 2

System.out.println(c);

LargeFactorial

BigDecimal a = **new** BigDecimal(1.0);

BigDecimal b = **new** BigDecimal(3);

BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);

System.out.println(c);

The `String` Class

❑ Constructing a String:

```
String message = "Welcome to Java";
String message = new String("Welcome to Java");
String s = new String();
```

❑ Obtaining String length and Retrieving Individual Characters in a string

❑ String Concatenation (concat)

❑ Substrings (substring(index), substring(start, end))

❑ Comparisons (equals, compareTo)

❑ String Conversions

❑ Finding a Character or a Substring in a String

❑ Conversions between Strings and Arrays

❑ Converting Characters and Numeric Values to Strings

Constructing Strings

String newString = new String(stringLiteral);

String message = new String("Welcome to Java");

Since strings are used frequently, Java provides a shorthand initializer for creating a string:

String message = "Welcome to Java";

Strings Are Immutable

A String object is immutable; its contents cannot be changed. Does the following code change the contents of the string?
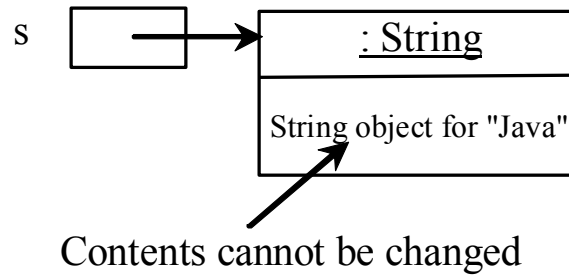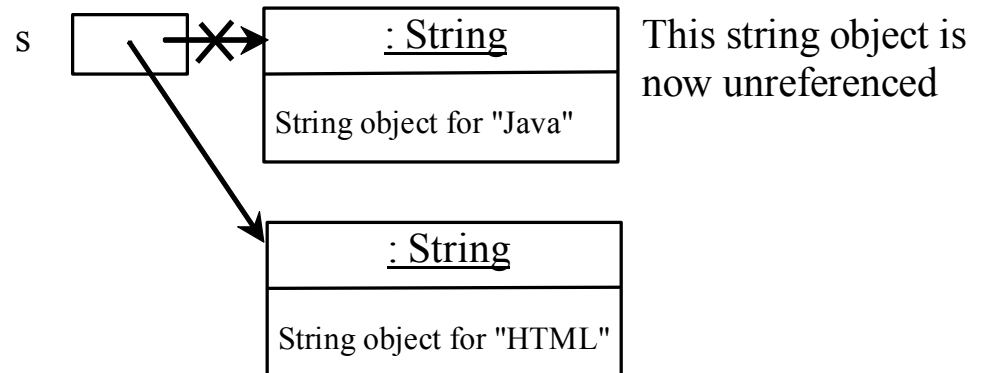
```
String s = "Java";
s = "HTML";
```

Trace Code

String s = "Java";
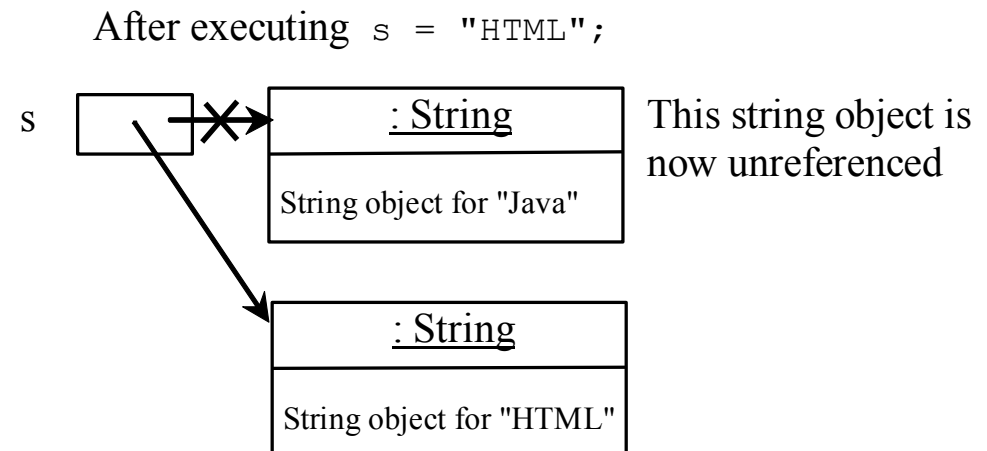
s = "HTML";

After executing `String s = "Java";`

s

: String

String object for "Java"

Contents cannot be changed

After executing `s = "HTML";`

s

: String

String object for "Java"

This string object is now unreferenced

: String

String object for "HTML"

Trace Code

String s = "Java";

s = "HTML";

After executing `String s = "Java";`

s → : String

String object for "Java"

Contents cannot be changed

After executing `s = "HTML";`

s → : String

String object for "Java"

This string object is now unreferenced

: String

String object for "HTML"
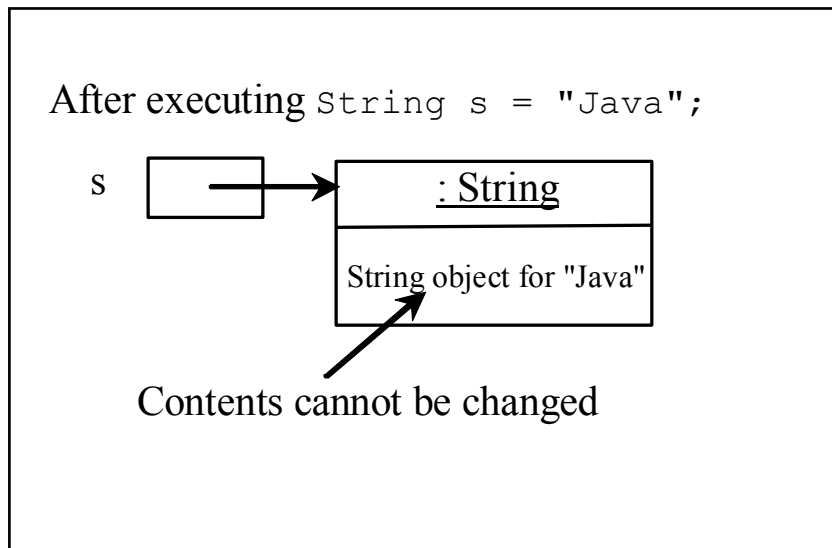
Interned Strings

Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a unique instance for string literals with the same character sequence. Such an instance is called *interned*. For example, the following statements:
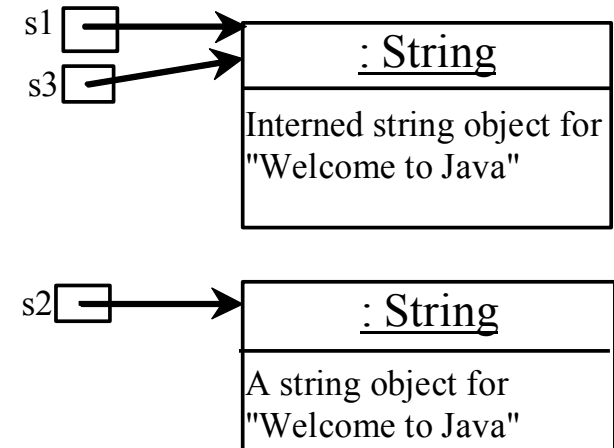
Examples

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```
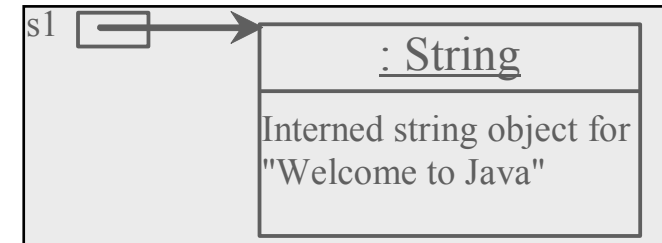
s1 □
s3 □

: String

Interned string object for "Welcome to Java"

s2 □

: String

A string object for "Welcome to Java"

display

s1 == s is false

s1 == s3 is true

A new object is created if you use the new operator.

If you use the string initializer, no new object is created if the interned object is already created.

36

Trace Code

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";
```
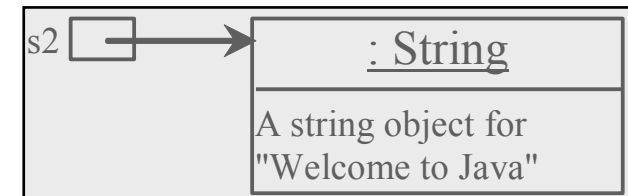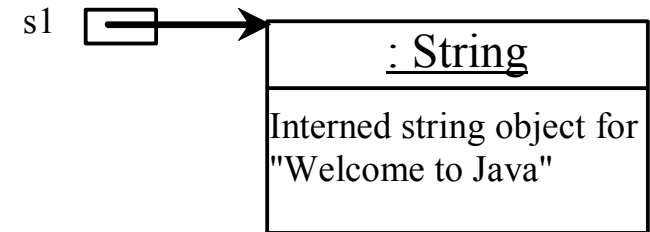
s1	
	: String
	Interned string object for "Welcome to Java"

Trace Code

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";
```

s1

: String

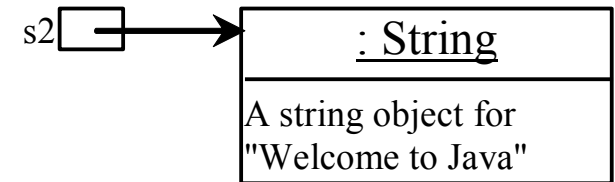Interned string object for
"Welcome to Java"

s2

: String

A string object for
"Welcome to Java"

Trace Code

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";
```
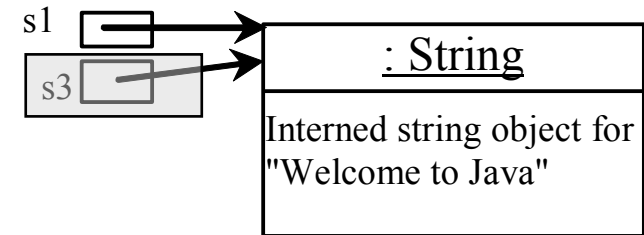
s1

s3

: String

Interned string object for "Welcome to Java"

s2

: String

A string object for "Welcome to Java"

39

Replacing and Splitting Strings

java.lang.String	
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching character in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replace all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

Examples

"Welcome".replace('e', 'A') returns a new string, WAlcomA.

"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.

"Welcome".replace("e", "AB") returns a new string, WABlcomAB.

"Welcome".replace("el", "AB") returns a new string, WABcome.

Splitting a String

```
String[] tokens = "Java#HTML#Perl".split("#", 0);
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

displays

Java HTML Perl

Matching, Replacing and Splitting by Patterns

You can match, replace, or split a string by specifying a pattern.
An extremely useful and powerful feature, commonly known
as *regular expression*.

Regular expression is complex to beginning students..

"Java".matches("Java");

"Java".equals("Java");

"Java is fun".matches("Java.*");

"Java is cool".matches("Java.*");

Matching, Replacing and Splitting by Patterns

The replaceAll, replaceFirst, and split methods can be used with a regular expression. For example, the following statement returns a new string that replaces $, +, or # in "a+b$#c" by the string NNN.

String s = "a+b$#c".replaceAll("[$+#]", "NNN");

System.out.println(s);

Here the regular expression [$+#] specifies a pattern that matches $, +, or #. So, the output is aNNNbNNNNNNc.

Matching, Replacing and Splitting by Patterns

The following statement splits the string into an array of strings
 delimited by some punctuation marks.

String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);

Regular Expressions

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. Regular expression is a powerful tool for string manipulations. You can use regular expressions for matching, replacing, and splitting strings.

Matching Strings

```
"Java".matches("Java");
"Java".equals("Java");

"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

Regular Expression Syntax

Regular Expression	Matches	Example
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab|cd)	ab or cd	ten matches t(en|im)
[abc]	a, b, or c	Java matches Ja[uvwx]a
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]
\d	a digit, same as [0-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java1 matches "[\\w]ava[\\w]"
\W	a non-word character	$Java matches "[\\W][\\w]ava"
\s	a whitespace character	"Java 2" matches "Java\\s2"
\S	a non-whitespace char	Java matches "[\\S]ava"
p*	zero or more occurrences of pattern p	aaaabb matches "a*bb" ababab matches "(ab)*"
p+	one or more occurrences of pattern p	a matches "a+b*" able matches "(ab)+.*"
p?	zero or one occurrence of pattern p	Java matches "J?Java" Java matches "J?ava"
p{n}	exactly n occurrences of pattern p	Java matches "Ja{1}.*" Java does not match ".{2}"
p{n,}	at least n occurrences of pattern p	aaaa matches "a{1,}" a does not match "a{2,}"
p{n,m}	between n and m occurrences (inclusive)	aaaa matches "a{1,9}" abb does not match "a{2,9}bb"

48

Replacing and Splitting Strings

java.lang.String	
+matches(regex: String): boolean	Returns true if this string matches the pattern.
+replaceAll(regex: String, replacement: String): String	Returns a new string that replaces all matching substrings with the replacement.
+replaceFirst(regex: String, replacement: String): String	Returns a new string that replaces the first matching substring with the replacement.
+split(regex: String): String[]	Returns an array of strings consisting of the substrings split by the matches.

Examples

TestToken

RegularExTester

Java provides the java.util.regex package for pattern matching with regular expressions.

Pattern Class − A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. Using its public static compile() methods will return a Pattern object.

Matcher Class − A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Matcher defines no public constructors. Obtaining a Matcher object by invoking the matcher() method on a Pattern object.

PatternSyntaxException − A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

RegexDemo

Convert Character and Numbers to Strings

The String class provides several static valueOf methods for converting a character, an array of characters, and numeric values to strings. These methods have the same name valueOf with different argument types char, char[], double, long, int, and float.

For example, to convert a double value to a string, use String.valueOf(5.44). The return value is string consists of characters '5', '.', '4', and '4'.

`StringBuilder` and `StringBuffer`

The `StringBuilder`/`StringBuffer` class is an alternative to the `String` class. In general, a StringBuilder/StringBuffer can be used wherever a string is used. StringBuilder/StringBuffer is more flexible than String. You can add, insert, or append new contents into a string buffer, whereas the value of a String object is fixed once the string is created.

`StringBuilder` Constructors

java.lang.StringBuilder	
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

Modifying Strings in the Builder

java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array to the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: *aPrimitiveType*): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

Examples

stringBuilder.append("Java");

stringBuilder.insert(11, "HTML and ");

stringBuilder.delete(8, 11) changes the builder to Welcome
 Java.

stringBuilder.deleteCharAt(8) changes the builder to
 Welcome o Java.

stringBuilder.reverse() changes the builder to avaJ ot
 emocleW.

stringBuilder.replace(11, 15, "HTML")
  changes the builder to Welcome to HTML.

stringBuilder.setCharAt(0, 'w') sets the builder to welcome
 to Java.

The toString, capacity, length, setLength, and charAt Methods

java.lang.StringBuilder	
+toString(): String	Returns a string object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at startIndex.
+substring(startIndex: int, endIndex: int): String	Returns a substring from startIndex to endIndex-1.
+trimToSize(): void	Reduces the storage size used for the string builder.

Problem: Checking Palindromes Ignoring Non-alphanumeric Characters

This example gives a program that counts the number of occurrence of each letter in a string. Assume the letters are not case-sensitive.

<u>PalindromeIgnoreNonAlphanumeric</u>