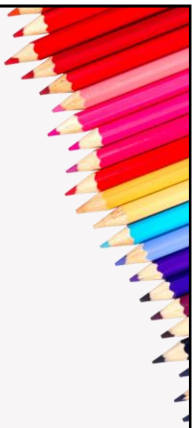
# Chapter 10
# *Thinking in Objects*



## Motivations

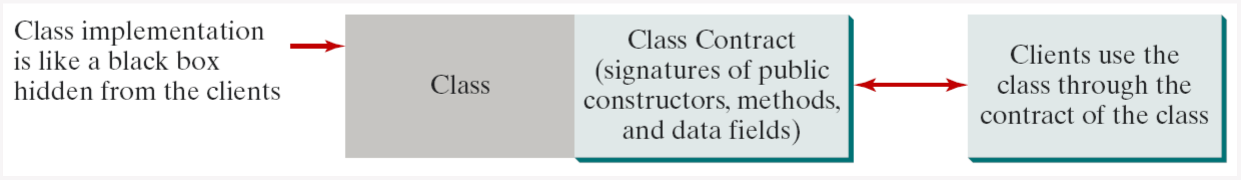
- You see the advantages of object-oriented programming from the preceding chapter. This chapter will demonstrate how to solve problems using the object-oriented paradigm.
- *The focus of this chapter is on class design and explores the differences between procedural programming and object-oriented programming.*

## Objectives

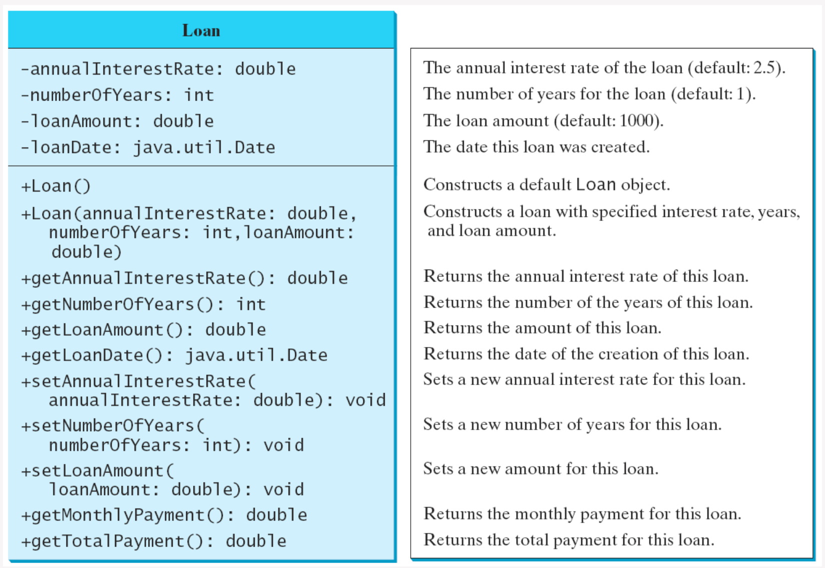
- To apply class abstraction to develop software (§10.2).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.3).
- To discover the relationships between classes (§10.4).
- To design programs using the object-oriented paradigm (§§10.5–10.6).
- To create objects 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).

## 10.2 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 constructors, methods, and data fields)
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: java.util.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(): java.util.Date` | Returns the date of the creation of this loan. |
| `+setAnnualInterestRate( annualInterestRate: double): void` | Sets a new annual interest rate for this loan. |
| `+setNumberOfYears( numberOfYears: int): void` | Sets a new number of years for this loan. |
| `+setLoanAmount( loanAmount: double): void` | Sets a new amount for this loan. |
| `+getMonthlyPayment(): double` | Returns the monthly payment for this loan. |
| `+getTotalPayment(): double` | Returns the total payment for this loan. |

Loan

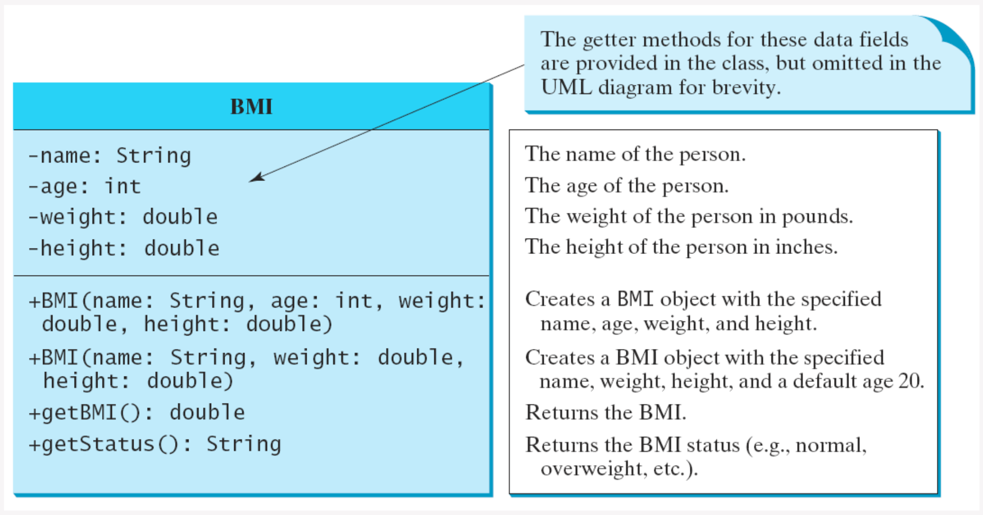
TestLoanClass

Run

## 10.3 Thinking in Objects

- 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.
- This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach.
- 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 getter 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

Run

7

## Thinking in Objects

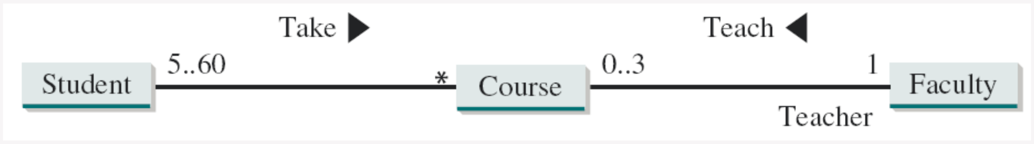
- *The procedural paradigm focuses on designing methods.*
- *The object-oriented paradigm couples data and methods together into objects.*
- *Software design using the object-oriented paradigm focuses on objects and operations on objects.*

## 10.4 Class Relationships

- To design classes, you need to explore the relationships among classes.
- The common relationships among classes are *association*, *aggregation*, *composition*, and *inheritance*.
    - This section explores association, aggregation, and composition.
    - The inheritance relationship will be introduced in the next chapter.

## Association

- *Association* is a general binary relationship that describes an activity between two classes.
- For example, a student taking a course is an association between the Student class and the Course class, and a faculty member teaching a course is an association between the Faculty class and the Course class.



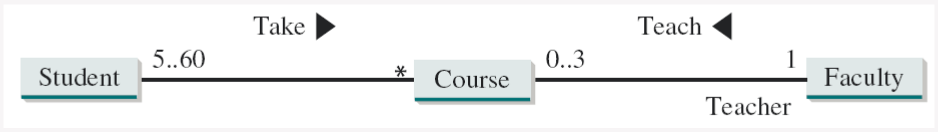
## Class Representation

- The association relations are implemented using data fields and methods in classes.

```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

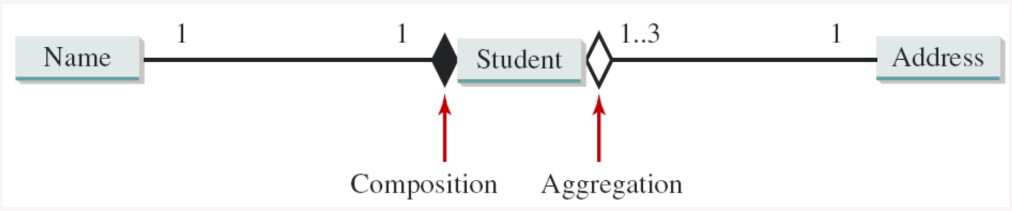
```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```



## Aggregation and Composition

- *Composition* is actually a special case of the aggregation 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.
- Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

```
public class Name {
  ...
}
```

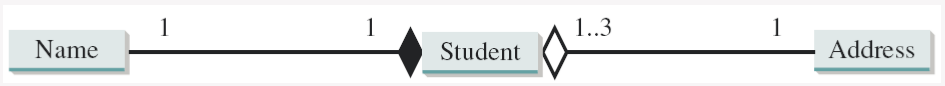
Aggregated class

```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

Aggregating class

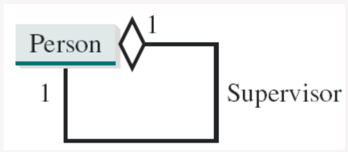
```
public class Address {
  ...
}
```

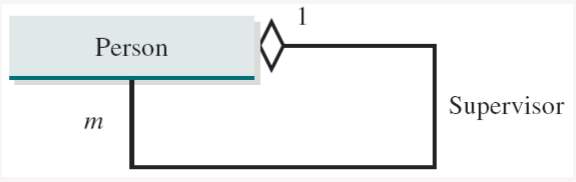
Aggregated class



## Aggregation Between Same Class

- Aggregation may exist between objects of the same class. For example, a person may have a supervisor.
- If a person can have several supervisors, you may use an array to store supervisors.

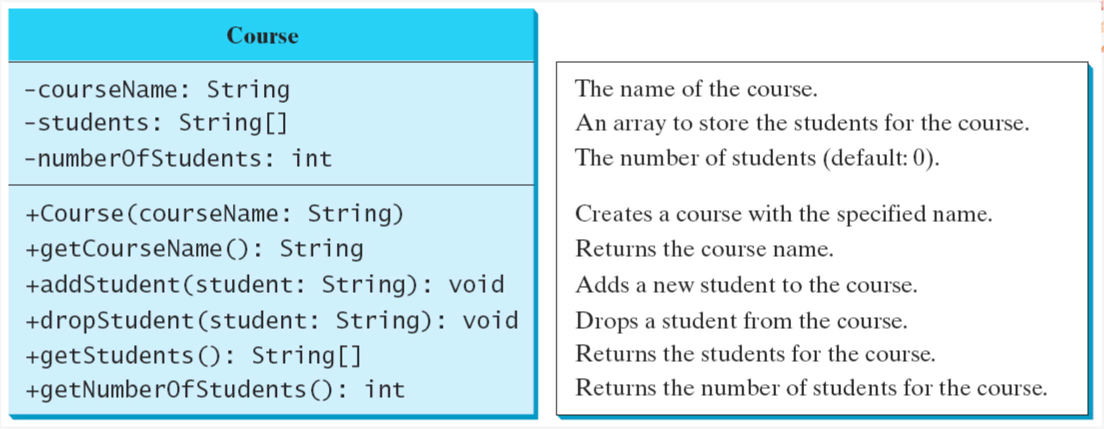




```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;

  ...
}
```

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

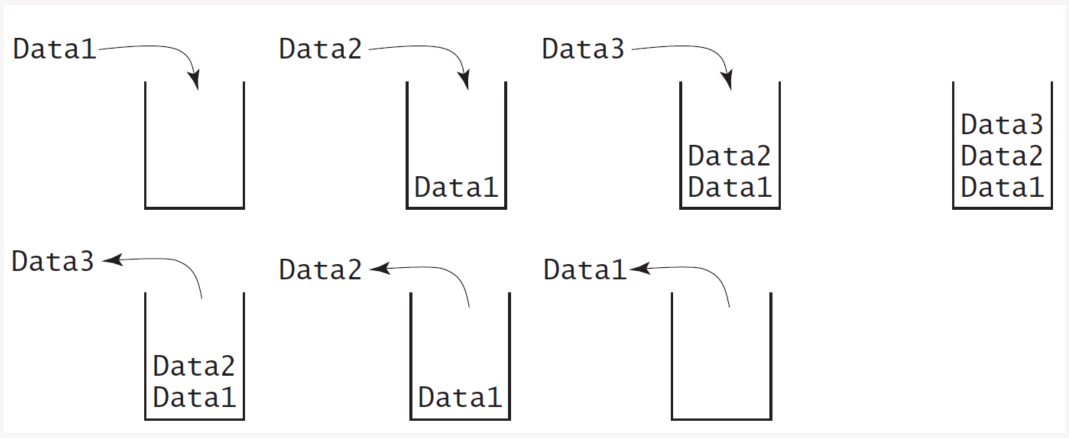
## Case Study: Designing The Course Class



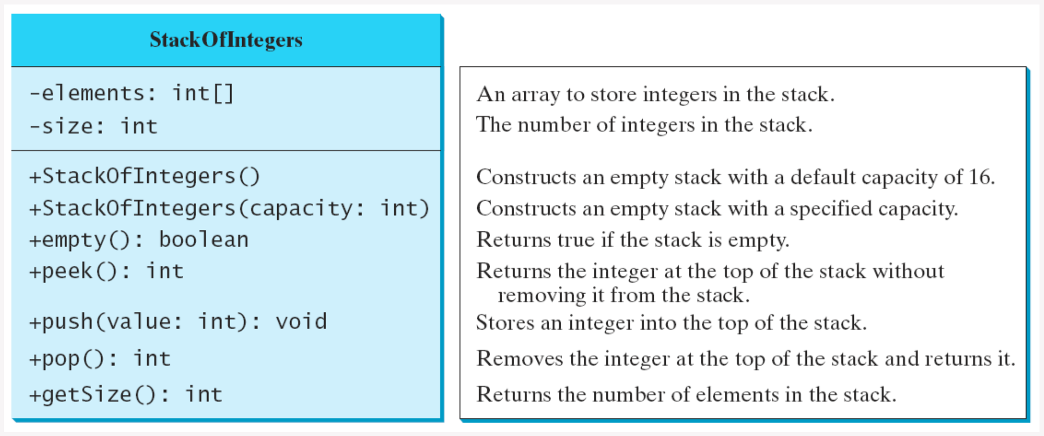
| Course | |
|---|---|
| -courseName: String | The name of the course. |
| -students: String[] | An array to store the students for the course. |
| -numberOfStudents: int | The number of students (default: 0). |
| +Course(courseName: String) | Creates a course with the specified name. |
| +getCourseName(): String | Returns the course name. |
| +addStudent(student: String): void | Adds a new student to the course. |
| +dropStudent(student: String): void | Drops a student from the course. |
| +getStudents(): String[] | Returns the students for the course. |
| +getNumberOfStudents(): int | Returns the number of students for the course. |

Course TestCourse Run

## Case Study: Designing a Class for Stacks



## The StackOfIntegers Class

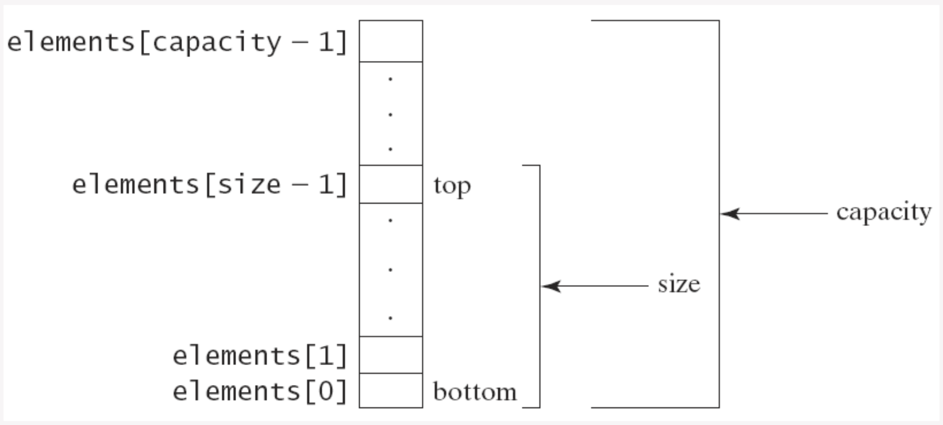


| StackOfIntegers | |
|---|---|
| -elements: int[] | An array to store integers in the stack. |
| -size: int | The number of integers in the stack. |
| +StackOfIntegers() | Constructs an empty stack with a default capacity of 16. |
| +StackOfIntegers(capacity: int) | Constructs an empty stack with a 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): void | 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

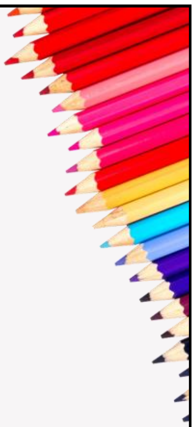
Run

## Implementing StackOfIntegers Class



StackOfIntegers

## 10.7 Processing Primitive Data Type Values as Objects

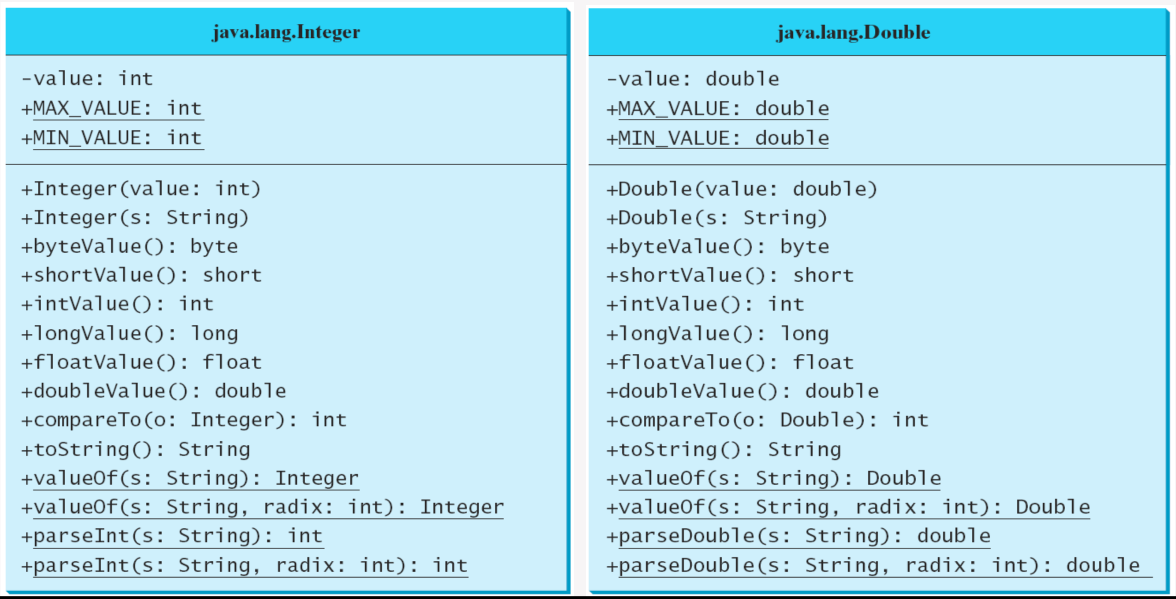


- *A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.*
- NOTE:

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.

- Boolean
- *Character*
- Short
- Byte
- *Integer*
- Long
- Float
- Double

## 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
+longValue(): 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
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double
```

## Numeric Wrapper Class Constructors and Constants

- 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:

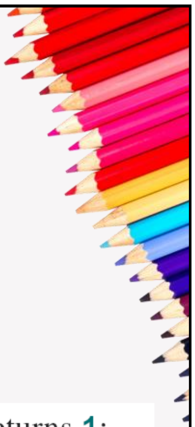
```
public Integer(int value)
public Integer(String s)
public Double(double
  value)
public Double(String s)
```

- 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.

## Conversion Methods

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

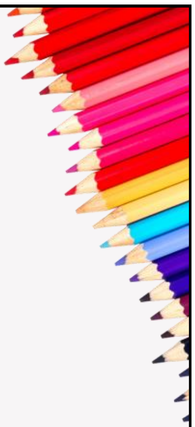
```
Double.valueOf(12.4).intValue() returns 12;
Integer.valueOf(12).doubleValue() returns 12.0;
```



## compareTo method

- The numeric wrapper classes contain the compareTo method for comparing two numbers and returns 1, 0, or -1, if this number is greater than, equal to, or less than the other number.
- For example,

```
Double.valueOf(12.4).compareTo(Double.valueOf(12.3)) returns 1;
Double.valueOf(12.3).compareTo(Double.valueOf(12.3)) returns 0;
Double.valueOf(12.3).compareTo(Double.valueOf(12.51)) returns -1;
```



## The Static valueOf 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");
```

24

## The Methods for Parsing Strings into Numbers

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

```
Integer.parseInt("11", 2) returns 3;
Integer.parseInt("12", 8) returns 10;
Integer.parseInt("13", 10) returns 13;
Integer.parseInt("1A", 16) returns 26;
```

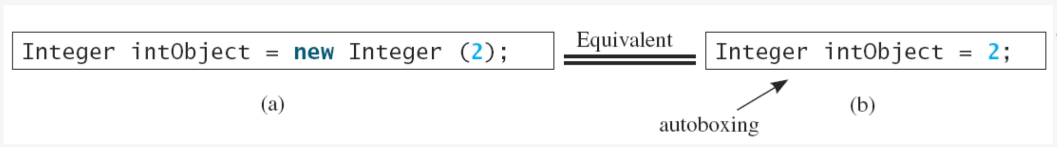
## The Methods for Parsing Strings into Numbers

```
// These two methods are in the Byte class
public static byte parseByte(String s)
public static byte parseByte(String s, int radix)

// These two methods are in the Short class
public static short parseShort(String s)
public static short parseShort(String s, int radix)

// These two methods are in the Integer class
public static int parseInt(String s)
public static int parseInt(String s, int radix)

// These two methods are in the Long class
public static long parseLong(String s)
public static long parseLong(String s, int radix)

// These two methods are in the Float class
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)

// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

## 10.8 Automatic Conversion between Primitive Types and Wrapper Class Types

- *A primitive type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context.*
- Converting a primitive value to a wrapper object is called *boxing*. *The reverse conversion is* called *unboxing*.
- *Java allows primitive types and wrapper classes to be converted automatically.*
  - The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value.This is called *autoboxing and autounboxing.*

## Autoboxing and Autounboxing



```
Integer intObject = new Integer (2);
```

```
Integer intObject = 2;
```

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

- In line 1, the primitive values 1,2, and 3 are automatically boxed into objects new Integer(1), new Integer(2), and new Integer(3).
- In line 2, the objects intArray[0], intArray[1], and intArray[2] are automatically unboxed into int values that are added together.

## 10.9 The BigInteger and BigDecimal Classes

- *The BigInteger and BigDecimal classes in the java.math package can be used to represent integers or decimal numbers of any size and precision.*
- If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes.
  - Both extend the Number class and implement the Comparable interface.
  - Both are immutable.

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);     // The output is 18446744073709551614.
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 output is 0.33333333333333333334.
```

LargeFactorial

Run

## 10.10 The String Class

- *A String object is immutable: Its content cannot be changed once the string is created.*
- 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

- To create a string from a string literal, use the syntax:

```
String newString = new String(stringLiteral);
```

- For example,

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

- Java treats a string literal as a String object. Thus, the following statement is valid:

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

- You can also create a string from an array of characters. For example,

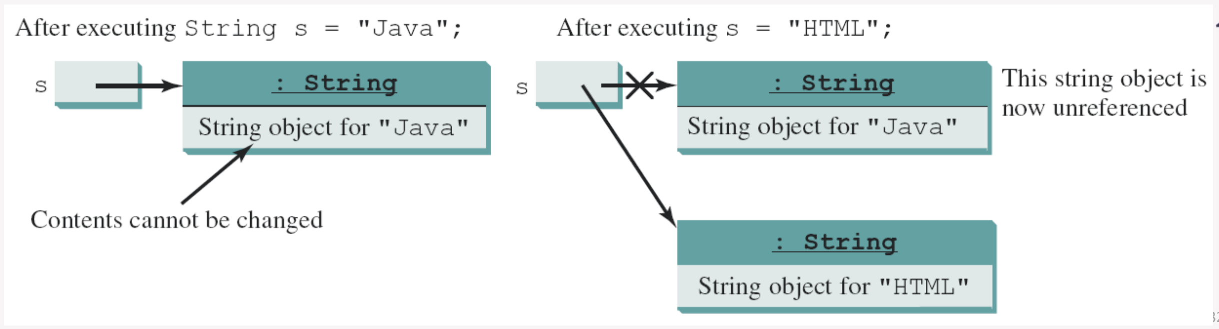
```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a',
  'y'};
String message = new String(charArray); // "Good Day"
```

31

## 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";
```

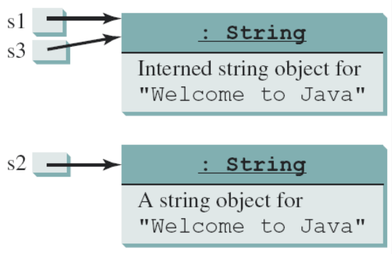

After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

32

## 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*.
- 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.

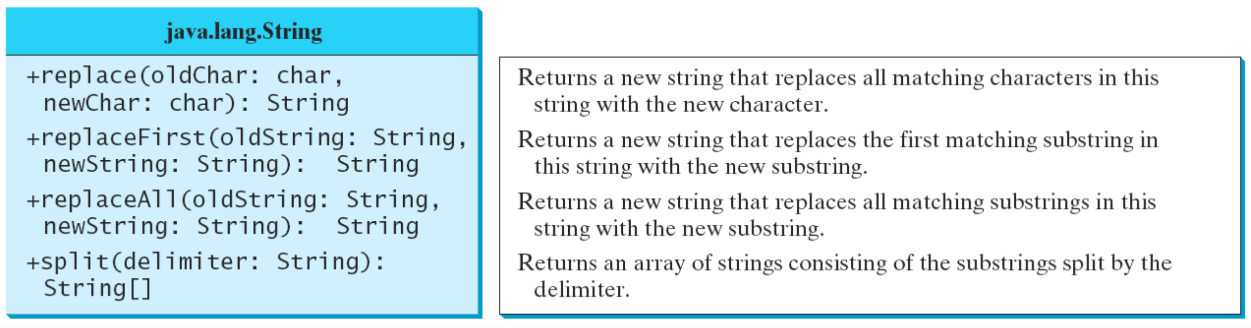
```
String s1 = "Welcome to Java";

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

String s3 = "Welcome to Java";

String s4 = new String("Welcome to Java");

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
System.out.println("s2 == s4 is " + (s2 == s4));
```


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

33

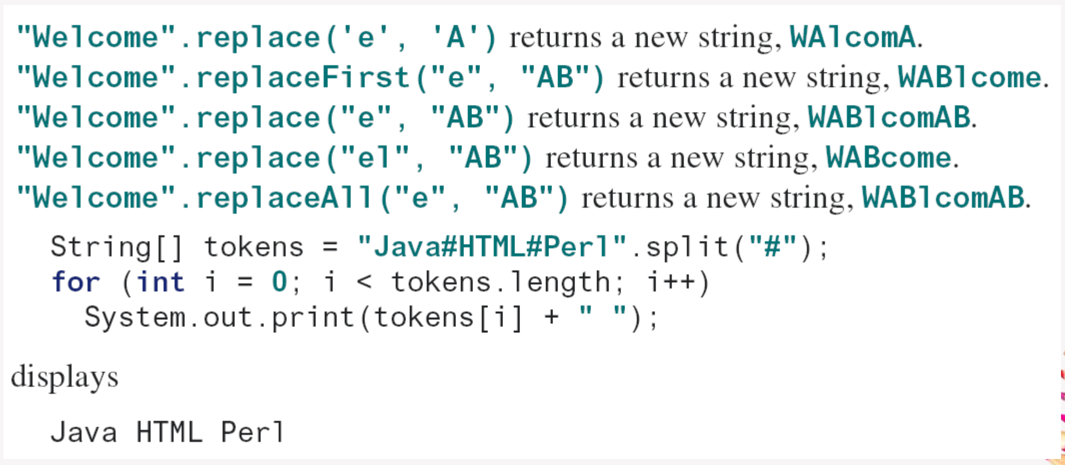
## Replacing and Splitting Strings

- The String class provides the methods for replacing and splitting strings.



| java.lang.String | |
|---|---|
| +replace(oldChar: char, newChar: char): String | Returns a new string that replaces all matching characters 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 replaces 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`.
`"Welcome".replaceAll("e", "AB")` returns a new string, `WABlcomAB`.

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

displays

```
Java HTML Perl
```

35

## Matching, Replacing and Splitting by Patterns

- You can match, replace, or split a string by specifying a pattern.
- A *regular expression (abbreviated regex*) is a string that describes a pattern for matching a set of strings.
  - Please refer to Supplement III.F, “Regular Expressions,” for further studies.
- Regular expression is a powerful tool for string manipulations.
- You can use regular expressions for matching, replacing, and splitting strings.

## 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"<br>ababab matches "(ab)*" |
| *p*+ | one or more occurrences of pattern *p* | a matches "a+b*"<br>able matches "(ab)+.*" |
| *p*? | zero or one occurrence of pattern *p* | Java matches "J?Java"<br>Java matches "J?ava" |
| *p*{n} | exactly n occurrences of pattern *p* | Java matches "Ja{1}.*"<br>Java does not match ".{2}" |
| *p*{n,} | at least n occurrences of pattern *p* | aaaa matches "a{1,}"<br>a does not match "a{2,}" |
| *p*{n,m} | between n and m occurrences (inclusive) | aaaa matches "a{1,9}"<br>abb does not match "a{2,9}bb" |

37

## Matching, 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. |
| `+split(regex: String, limit: int): String[]` | Same as the preceding split method except that the limit parameter controls the number of times the pattern is applied. |

38

## matches(regex)

- **matches** method can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to **true:**

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

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

"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

- The regular expression **Java.*** describes a string pattern that begins with Java followed by *any zero or more characters.*
- **\\d** represents a single digit, and **\\d{3}** represents three digits.

## replaceAll(regex)

- 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.

40

## split(regex)

- 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]);
```

In this example, the regular expression `[.,:;?]` specifies a pattern that matches `.`, `,`, `:`, `;`, or `?`. Each of these characters is a delimiter for splitting the string. Thus, the string is split into `Java`, `C`, `C#`, and `C++`, which are stored in array `tokens`.

## Conversion between Strings and Arrays

- Strings are not arrays, but a string can be converted into an array, and vice versa.
- To convert a string into an array of characters, use the **toCharArray** method.
- You can also use the **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin**) method to copy a substring of the string from index srcBegin to index srcEnd-1 into a character array dst starting from index dstBegin.
- To convert an array of characters into a string, use the **String(char[])** constructor or the **valueOf(char[])** method.

## Strings to Arrays

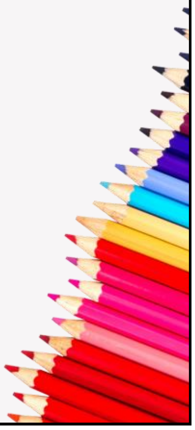
- The following statement converts the string **Java** to an array.

```
char[] chars = "Java".toCharArray();
```

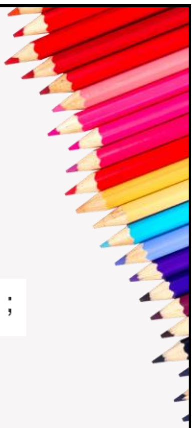
  - Thus, chars[0] is J, chars[1] is a, chars[2] is v, and chars[3] is a.
- The following code copies a substring **"3720"** in **"CS3720"** from index 2 to index 6-1 into the character array dst starting from index 4.

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

  - Thus, dst becomes {'J', 'A', 'V', 'A', '3', '7', '2', '0'}.



## Arrays to Strings



- The following statement constructs a string from an array using the **String** constructor.

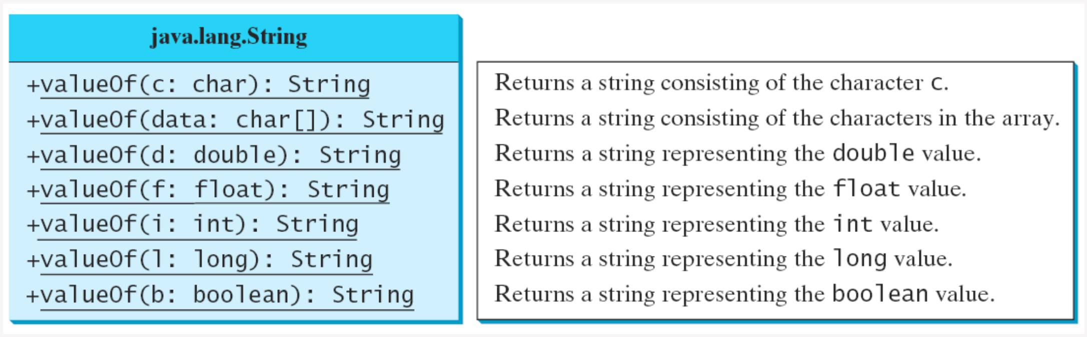
```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

- The following statement constructs a string from an array using the **valueOf** method.

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

## Converting Characters and Numeric Values to Strings

- String to numeric value:
  - **Double.parseDouble(str), Integer.parseInt(str)**
- Primitive values to Strings
  - string concatenating operator **(+)** or the overloaded static **valueOf** method.



| java.lang.String | |
|---|---|
| +valueOf(c: char): String | Returns a string consisting of the character c. |
| +valueOf(data: char[]): String | Returns a string consisting of the characters in the array. |
| +valueOf(d: double): String | Returns a string representing the double value. |
| +valueOf(f: float): String | Returns a string representing the float value. |
| +valueOf(i: int): String | Returns a string representing the int value. |
| +valueOf(l: long): String | Returns a string representing the long value. |
| +valueOf(b: boolean): String | Returns a string representing the boolean value. |

## Formatting Strings

- The String class contains the static **format** method to return a formatted string. The syntax to invoke this method is:

```
String.format(format, item1, item2, ..., itemk)
```

- For example,

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
```

- Note that

```
System.out.printf(format, item1, item2, ..., itemk);
```

is equivalent to

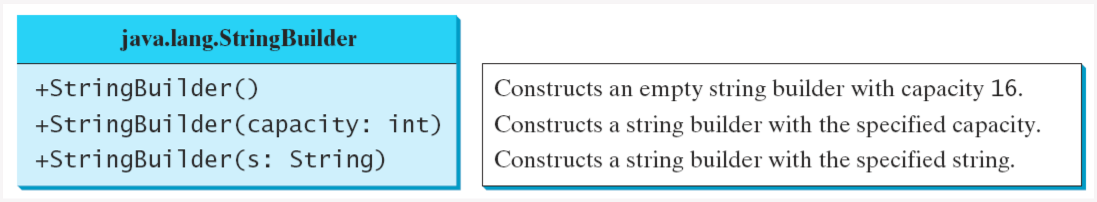
```
System.out.print(String.format(format, item1, item2, ...,
 itemk));
```

## 10.11 The StringBuilder and StringBuffer Classes

- 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.
- The `StringBuilder` class is similar to `StringBuffer` except that the methods for modifying the buffer in `StringBuffer` are *synchronized,* which means that only one task is allowed to execute the methods.

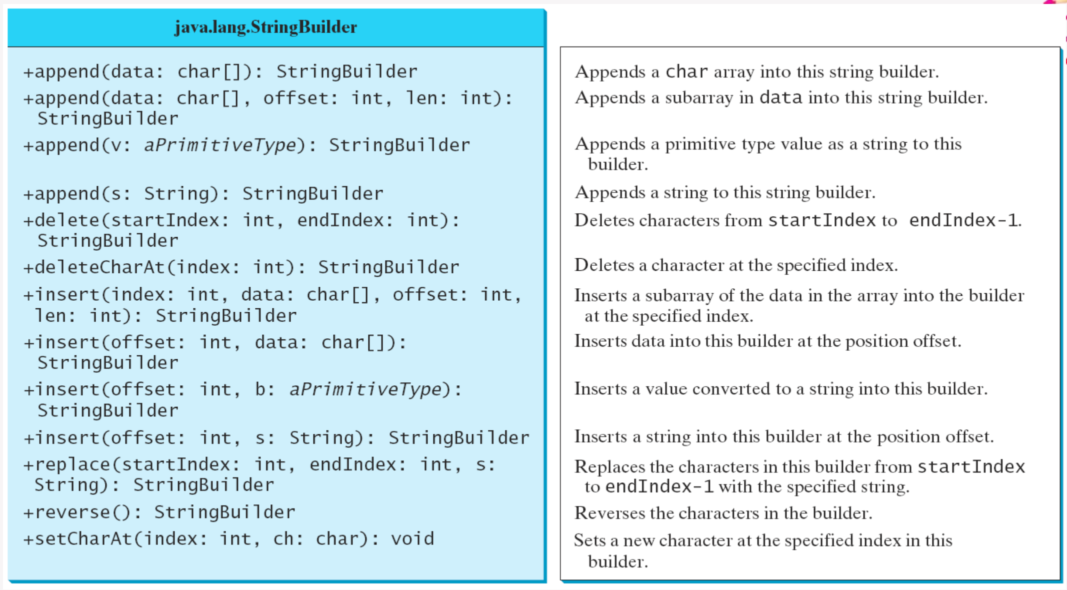
## StringBuilder Constructors

- The StringBuilder class contains the constructors for creating instances of StringBuilder.



| 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 StringBuilder



| 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-1`. |
| `+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 into 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-1` 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. |

49

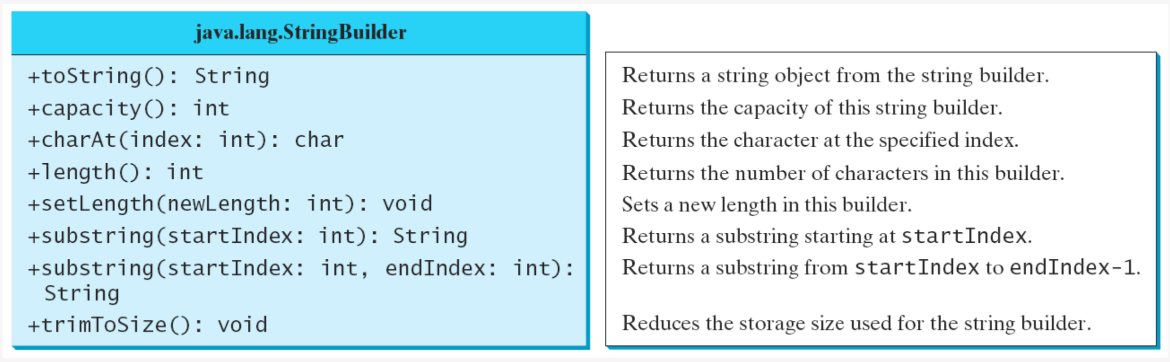
## Examples

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

`stringBuilder.insert(11, "HTML and ");` is `Welcome to HTML and Java`.

`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. |

51

## Case Study: 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.

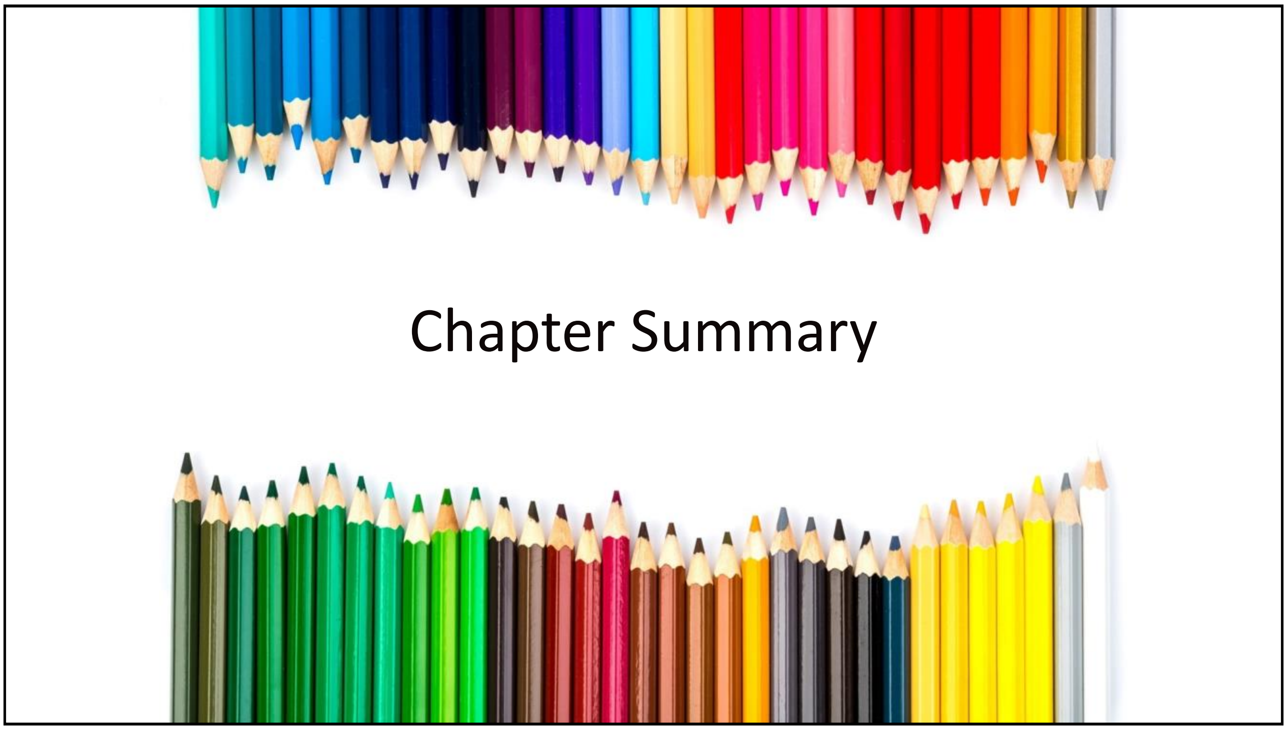
```
Enter a string: ab<c>cb?a ↵Enter
Ignoring nonalphanumeric characters,
is ab<c>cb?a a palindrome? true
```

```
Enter a string: abcc><?cab ↵Enter
Ignoring nonalphanumeric characters,
is abcc><?cab a palindrome? false
```

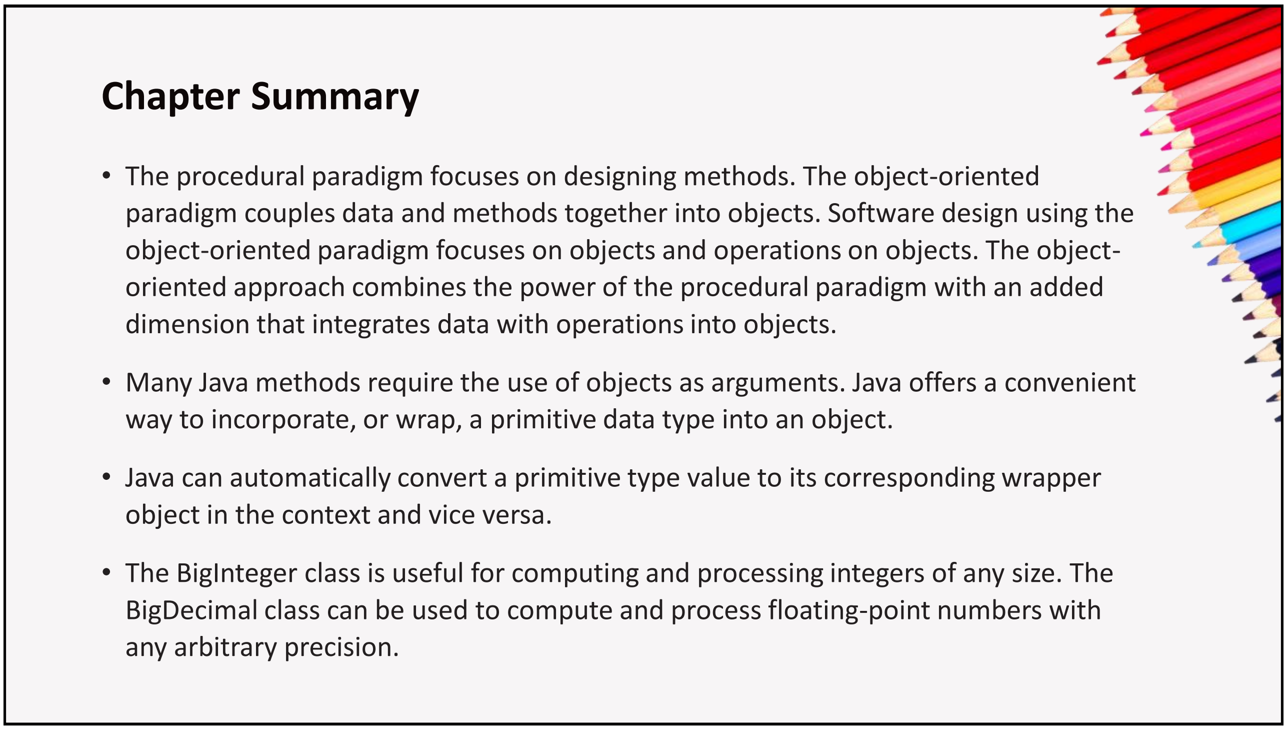
PalindromeIgnoreNonAlphanumeric

Run

52



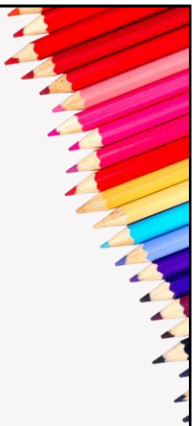
## Chapter Summary



## Chapter Summary

- The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.
- Many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object.
- Java can automatically convert a primitive type value to its corresponding wrapper object in the context and vice versa.
- The BigInteger class is useful for computing and processing integers of any size. The BigDecimal class can be used to compute and process floating-point numbers with any arbitrary precision.

## Chapter Summary



- A String object is immutable; its contents cannot be changed. To improve efficiency and save memory, the JVM stores two literal strings that have the same character sequence in a unique object. This unique object is called an interned string object.
- A regular expression (abbreviated regex) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern.
- The StringBuilder and StringBuffer classes can be used to replace the String class. The String object is immutable, but you can add, insert, or append new contents into StringBuilder and StringBuffer objects. Use String if the string contents do not require any change, and use StringBuilder or StringBuffer if they might change.

## *Programming Exercises*

***1, 2, 3, 4, 5, 9, 10, 11,***

***12, 14, 18, 22, 26, 27***

56