Thinking in Objects

CSE 114: Introduction to Object-Oriented Programming

Paul Fodor

Stony Brook University

http://www.cs.stonybrook.edu/~cse114

Contents

- Immutable Objects and Classes
- Scope of Variables and Default values
- The **`this`** Keyword
  - Calling Constructors within Constructors
  - Refer to "hidden" data fields
- Class Abstraction and Encapsulation
  - Designing and implementing the Loan Class
  - Designing and implementing the BMI Class
  - Designing and implementing the Course Class
  - Designing and implementing the StackOfIntegers Class
- The String Class in detail, Regular Expressions and Command-Line Arguments
- StringBuilder and StringBuffer
- The Character Class
- Designing Classes

Immutable Objects and Classes

- ***Immutable object***: the contents of an object cannot be changed once the object is created
  - its class is called an ***immutable class***
    - Example immutable class: no set method in the Circle class:

```
public class Circle{
    private double radius;
    public Circle() { }
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
}
```

    - radius is private and cannot be changed without a set method

What Class is Immutable?

- A class with all private data fields and without mutators is not necessarily immutable!
- An ***immutable class***:
  1. It must mark all data fields `private`!
  2. Provide no mutator (set) methods!
  3. Provide no accessor methods that would return a reference to a mutable data field object!

Example mutable

```
public class Student {
  private int id;
  private BirthDate birthDate;
  public Student(int ssn, int year,
         int month, int day) {
    id = ssn;
    birthDate =
         new BirthDate(year, month, day);
  }
  public int getId() {
    return id;
  }
  public BirthDate getBirthDate() {
    return birthDate;
  }
}
```

```
public class BirthDate {
  private int year;
  private int month;
  private int day;

  public BirthDate(int newYear,
       int newMonth, int newDay) {
    year = newYear;
    month = newMonth;
    day = newDay;
  }
  public void setYear(int newYear){
    year = newYear;
  }
  public int getYear() {
    return year;
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, 2005, 1, 1);
    student.getBirthDate().setYear(2050);
    // Now the student birth year is changed:
    System.out.println(student.getBirthDate().getYear()); // 2050
  }
}
```

Scope of Variables and Default values

- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable
  - Also a local variable must be **initialized** explicitly before it can be used.

vs.

- Data Field Variables can be declared anywhere inside a class
  - The scope of instance and static variables is the entire class!
  - Initialized with default values.

The `this` Keyword

- The **`this`** keyword is the name of a reference that refers to an object itself
- Common uses of the **`this`** keyword:
  1. Reference a class's "***hidden***" *data fields*.
  2. To enable a constructor to **invoke another constructor** of the same class as the first statement in the constructor.

Reference the Hidden Data Fields

```
public class Foo {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    Foo.k = k;
  }
}
```

```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

Calling Overloaded Constructors

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

this must be explicitly used to reference the data field radius of the object being constructed

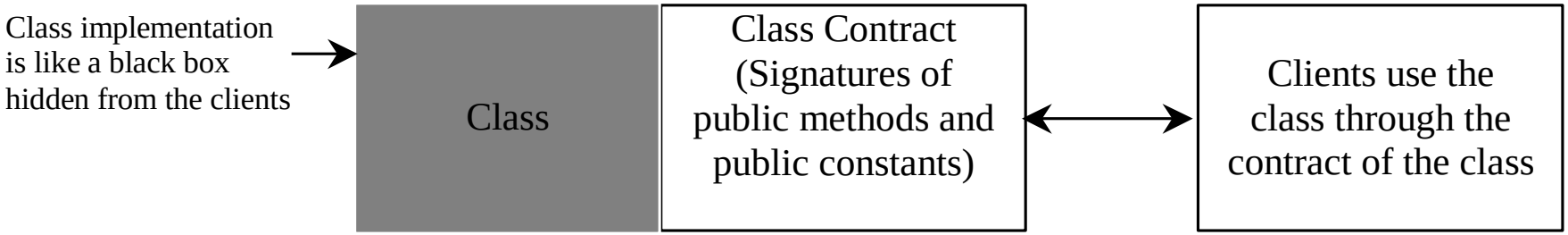
this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

Class Abstraction and Encapsulation

Class Abstraction = separate class implementation from the use of the class (API)

- The creator of the class provides a description of the class and let the user know how the class can be used.
- The user does not need to know how the class is implemented: it is ***encapsulated*** (private fields and only the public API is used).

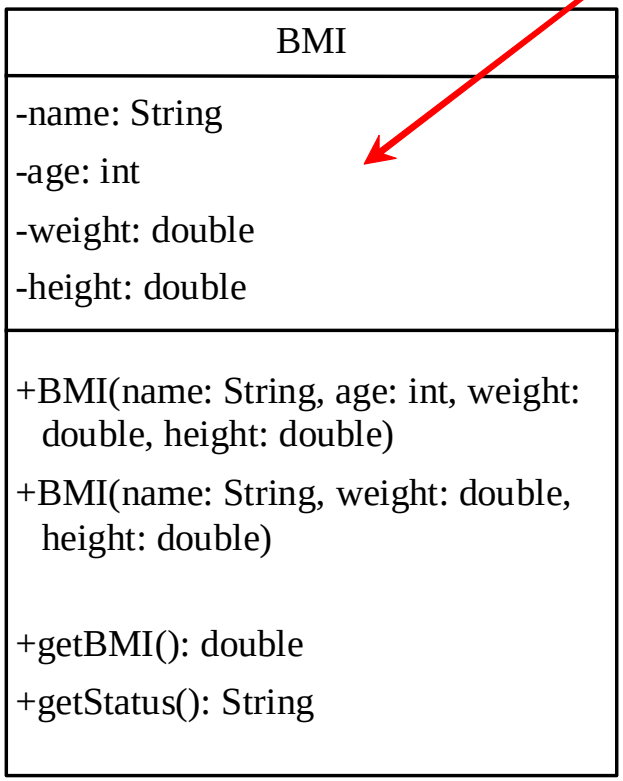


Designing the Loan Class

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.

```
public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;
  public Loan() {
    this(2.5, 1, 1000);
  }
  public Loan(double annualInterestRate, int numberOfYears,
      double loanAmount) {
    this.annualInterestRate = annualInterestRate;
    this.numberOfYears = numberOfYears;
    this.loanAmount = loanAmount;
    loanDate = new java.util.Date();
  }
  public double getMonthlyPayment() {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
      (Math.pow(1 / (1 + monthlyInterestRate), numberOfYears * 12)));
    return monthlyPayment;
  }
  public double getTotalPayment() {
    double totalPayment = getMonthlyPayment() * numberOfYears * 12;
    return totalPayment;
  } ...
}
```

The BMI Class



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

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.)

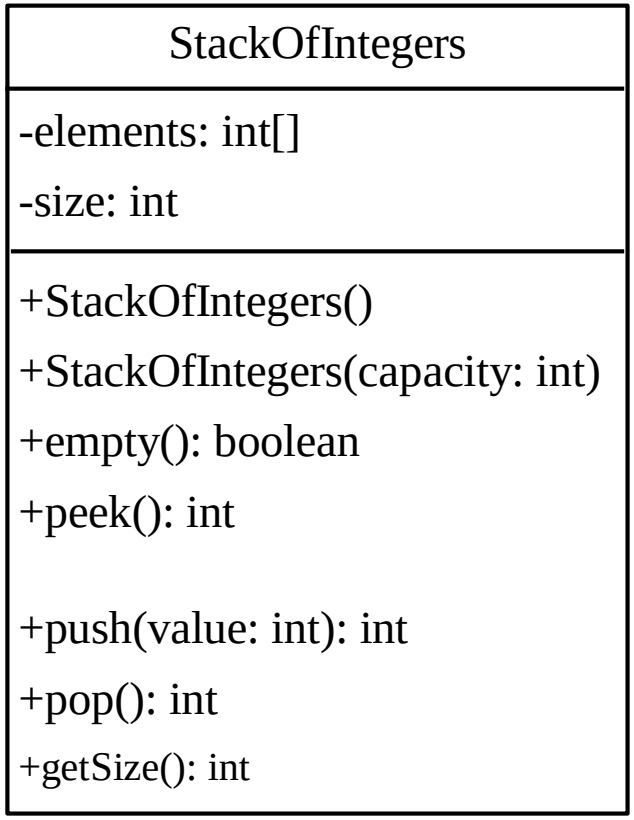
```
public class BMI {
  private String name;
  private int age;
  private double weight; // in pounds
  private double height; // in inches
  public static final double KILOGRAMS_PER_POUND = 0.45359237;
  public static final double METERS_PER_INCH = 0.0254;
  public BMI(String name, int age, double weight, double height) {
 this.name = name; this.age = age; this.weight = weight; this.height = height;
  }
  public double getBMI() {
    double bmi = weight * KILOGRAMS_PER_POUND /
     ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
    return Math.round(bmi * 100) / 100.0;
  }
  public String getStatus() {
    double bmi = getBMI();
    if (bmi < 16) return "seriously underweight";
    else if (bmi < 18) return "underweight";
    else if (bmi < 24) return "normal weight";
    else if (bmi < 29) return "over weight";
    else if (bmi < 35) return "seriously over weight";
    else return "gravely over weight";
  }
  public String getName() {    return name;  }
  public int getAge() {    return age;  }
  public double getWeight() {    return weight;  }
  public double getHeight() {    return height;  }
}
```

Example: The Course Class

Course	
-name: String	The name of the course.
-students: String[]	The students who take the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(name: String)	Creates a Course with the specified name.
+getName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course list.
+getStudents(): String[]	Returns the students for the course.
+getNumberOfStudents(): int	Returns the number of students for the course.

```
public class Course {
  private String courseName;
  private String[] students = new String[100];
  private int numberOfStudents;
  public Course(String courseName) {
    this.courseName = courseName;
  }
  public void addStudent(String student) {
    if(numberOfStudents >= students.length){
       String[] temp = new String[students.length * 2];
       System.arrayCopy(students, 0, temp, 0, students.length);
       students = temp;
    }
    students[numberOfStudents++] = student;
  }
  public String[] getStudents() {
    return students;
  }
  public int getNumberOfStudents() {
    return numberOfStudents;
  }
  public String getCourseName() {
    return courseName;
  }
}
```

Example: The `StackOfIntegers` Class



An array to store integers in the stack.

The number of integers in the stack.

Constructs an empty stack with a default capacity of 16.

Constructs an empty stack with a specified capacity.

Returns true if the stack is empty.

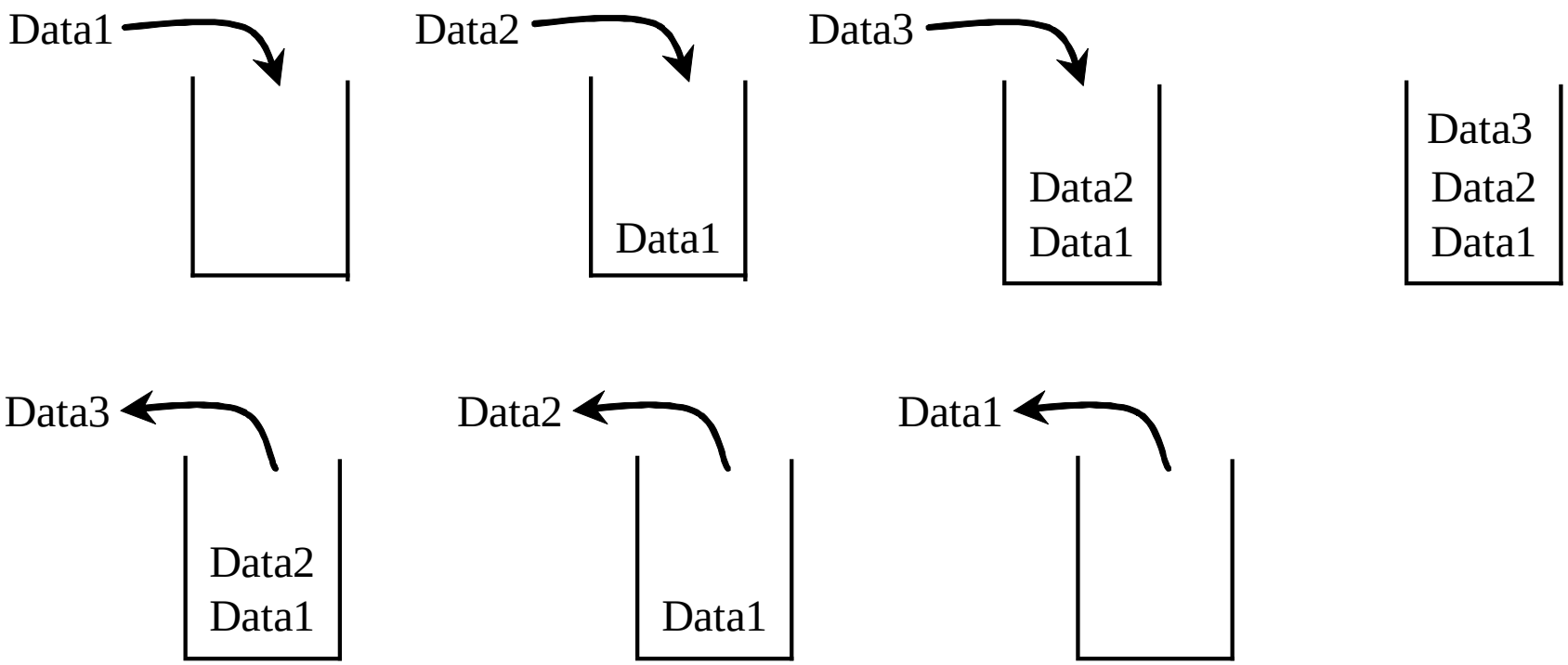
Returns the integer at the top of the stack without removing it from the stack.

Stores an integer into the top of the stack.

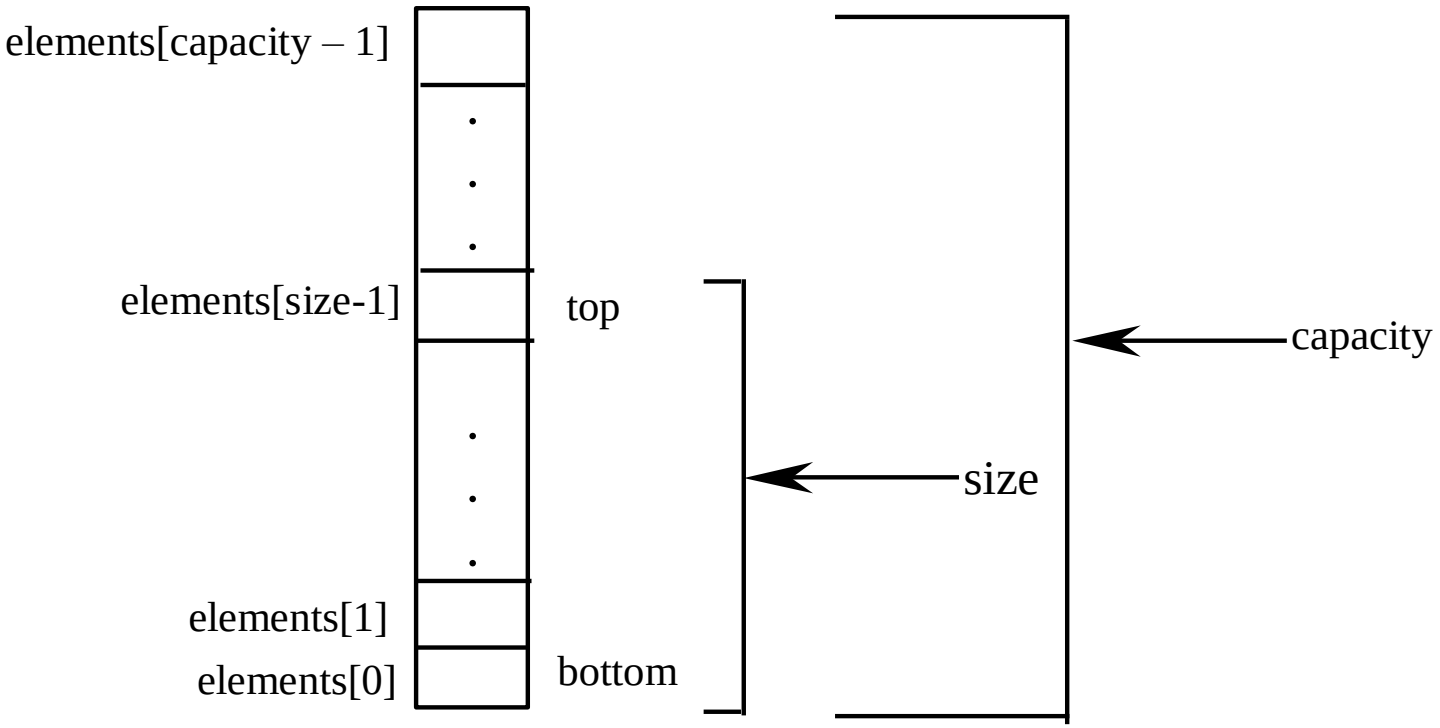
Removes the integer at the top of the stack and returns it.

Returns the number of elements in the stack.

Designing the `StackOfIntegers` Class



Implementing `StackOfIntegers` Class



```
public class StackOfIntegers {
  private int[] elements;
  private int size;
  public static final int DEFAULT_CAPACITY = 16;
  public StackOfIntegers() {
    this(DEFAULT_CAPACITY);
  }
  public StackOfIntegers(int capacity) {
    elements = new int[capacity];
  }
  public void push(int value) {
    if (size >= elements.length) {
      int[] temp = new int[elements.length * 2];
      System.arraycopy(elements, 0, temp, 0, elements.length);
      elements = temp;
    }
    elements[size++] = value;
  }
  public int pop() {
    return elements[--size];
  }
  public int peek() {
    return elements[size - 1];
  }
  public int getSize() {
    return size;
  }}
```

The String class in detail

- Remember the String class discussed at the beginning of the semester?
  - Constructing Strings:

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

    - Example:

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

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

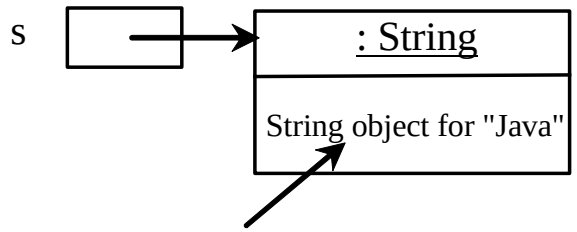
```
String message = "Welcome to Java";
                 // an Interned String
```

Strings Are Immutable

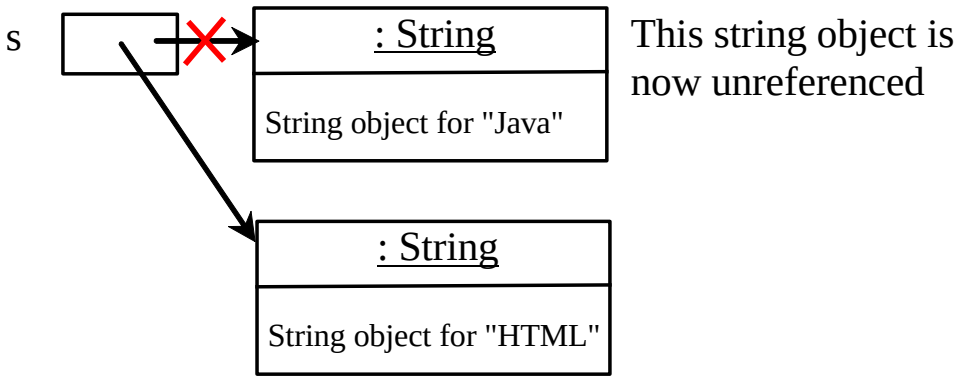
- A String object is immutable; its contents cannot be changed

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

After executing `String s = "Java";`



After executing `s = "HTML";`



Interned Strings

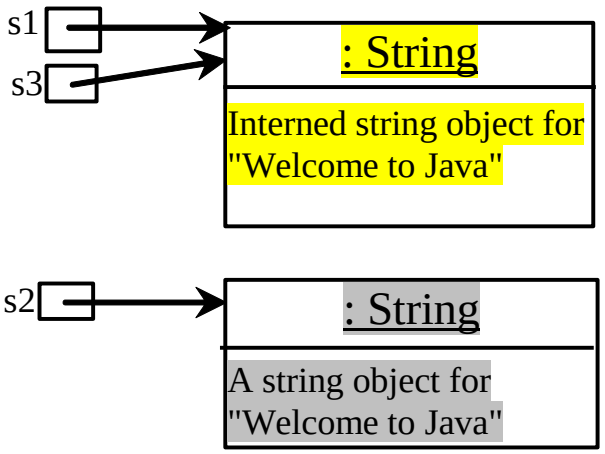
- ***String interning*** is a method of storing only one copy of each distinct compile-time constant/explicit string in the source code stored in a string intern pool (e.g., s1 and s3).
- If we use the **new** operator, then a new object is created in heap (e.g., s2).

```
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));
```



displays: `s1 == s3` is true

`s1 == s2` is false

The `String` Class methods

- Compare strings (equals, compareTo)
- Obtaining String length
- Retrieving Individual Characters in a string
- String Concatenation (concat)
- Substrings (substring(index), substring(start, end))
- Finding a Character or a Substring in a String
- Matching, Replacing and Splitting by Patterns
- Converting Characters and Numeric Values to Strings
- Command-Line Parameters

String Comparisons

- `equals(Object object)`:

```
String s1 = new String("Welcome");
String s2 = "Welcome";

if (s1.equals(s2)){ // true
  // s1 and s2 have the same contents
}

if (s1 == s2) { // false
  // s1 and s2 have different references
}
```

String Comparisons

- `equals(Object object)`:

```
String s1 = "Welcome";
String s2 = "Welcome";

if (s1.equals(s2)){ // true
  // s1 and s2 have the same contents
}

if (s1 == s2) { // true
  // s1 and s2 have the same reference
}
```

String Comparisons

- `compareTo(Object object):`

```
String s1 = new String("Welcome");
String s2 = new String("Welcome");

if (s1.compareTo(s2) > 0) {
  // s1 is greater than s2
}else if (s1.compareTo(s2) == 0) {
  // s1 and s2 have the same contents
}else{
  // s1 is less than s2
}
```

String Comparisons

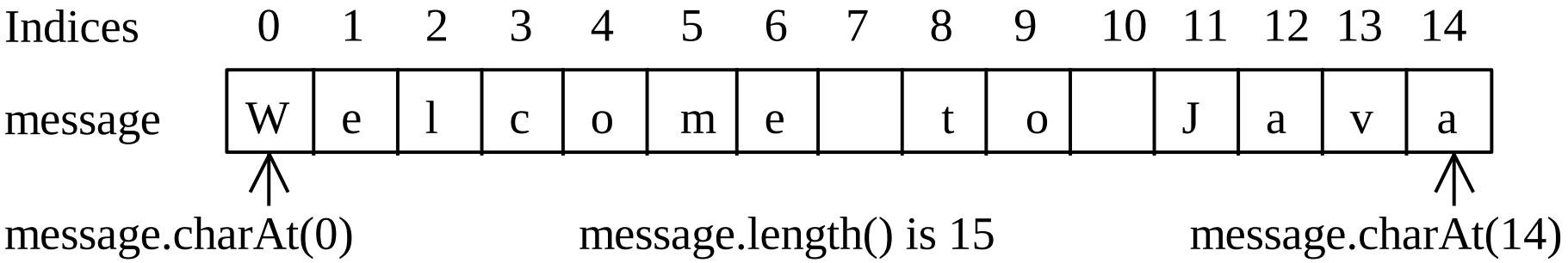
java.lang.String	
+equals(s1: String): boolean	Returns true if this string is equal to string s1.
+equalsIgnoreCase(s1: String): boolean	Returns true if this string is equal to string s1 case-insensitive.
+compareTo(s1: String): int	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.
+compareToIgnoreCase(s1: String): int	Same as compareTo except that the comparison is case-insensitive.
+regionMatches(toffset: int, s1: String, offset: int, len: int): boolean	Returns true if the specified subregion of this string exactly matches the specified subregion in string s1.
+regionMatches(ignoreCase: boolean, toffset: int, s1: String, offset: int, len: int): boolean	Same as the preceding method except that you can specify whether the match is case-sensitive.
+startsWith(prefix: String): boolean	Returns true if this string starts with the specified prefix.
+endsWith(suffix: String): boolean	Returns true if this string ends with the specified suffix.

String Length, Characters, and Combining Strings

java.lang.String	
+length(): int	Returns the number of characters in this string.
+charAt(index: int): char	Returns the character at the specified index from this string.
+concat(s1: String): String	Returns a new string that concatenate this string with string s1.

Retrieving Individual Characters in a String

- Use **`message.charAt(index)`→ `char`**
- Index starts from 0



Extracting Substrings

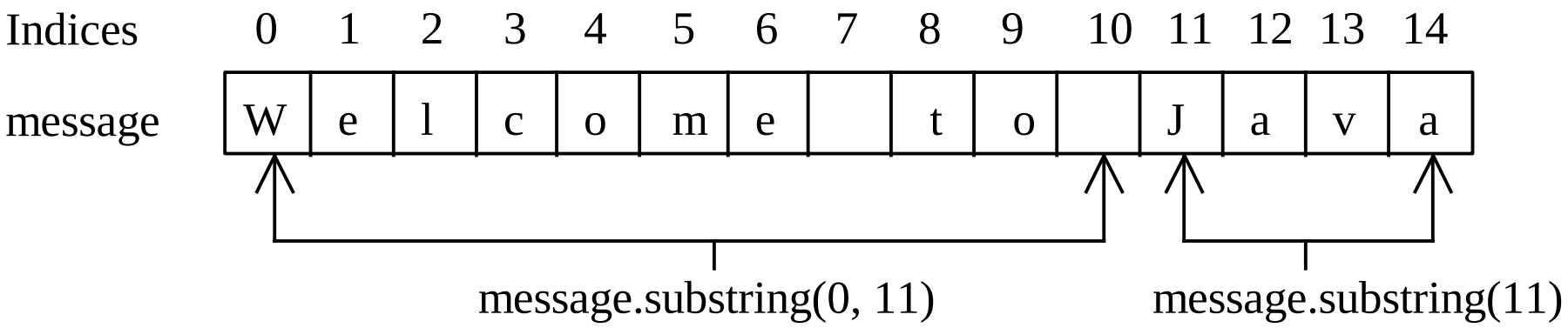
java.lang.String
+subString(beginIndex: int): String +subString(beginIndex: int, endIndex: int): String

Returns this string's substring that begins with the character at the specified beginIndex and extends to the end of the string.

Returns this string's substring that begins at the specified beginIndex and extends to the character at index endIndex – 1. Note that the character at endIndex is not part of the substring.

Extracting Substrings

```
String s1 = "Welcome to Java";
String s2 = s1.substring(0, 11) + "HTML";
```

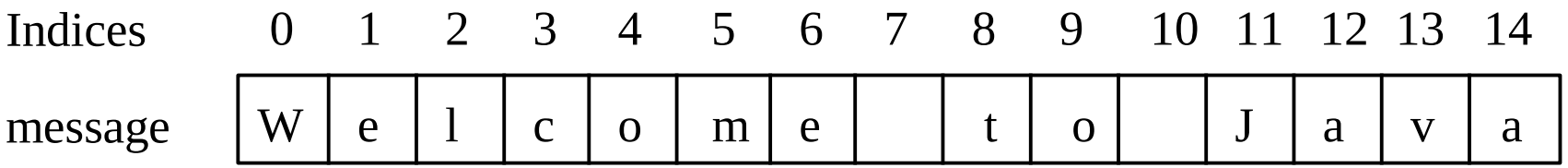


s2 will be "Welcome to HTML"

Finding a Character or a Substring in a String

java.lang.String	
+indexOf(ch: char): int	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
+indexOf(ch: char, fromIndex: int): int	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
+indexOf(s: String): int	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
+indexOf(s: String, fromIndex: int): int	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
+lastIndexOf(ch: int): int	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
+lastIndexOf(ch: int, fromIndex: int): int	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
+lastIndexOf(s: String): int	Returns the index of the last occurrence of string s. Returns -1 if not matched.
+lastIndexOf(s: String, fromIndex: int): int	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

Finding a Character or a Substring in a String




```
"Welcome to Java".indexOf('W') returns 0.
"Welcome to Java".indexOf('x') returns -1.
"Welcome to Java".indexOf('o', 5) returns 9.
"Welcome to Java".indexOf("come") returns 3.
"Welcome to Java".indexOf("Java", 5) returns 11.
"Welcome to Java".indexOf("java", 5) returns -1.
"Welcome to Java".lastIndexOf('a') returns 14.
```

Converting, Replacing, and Splitting Strings

java.lang.String	
+toLowerCase(): String	Returns a new string with all characters converted to lowercase.
+toUpperCase(): String	Returns a new string with all characters converted to uppercase.
+trim(): String	Returns a new string with blank characters trimmed on both sides.
+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".toLowerCase() returns a new string, "welcome".

"Welcome".toUpperCase() returns a new string, "WELCOME".

" Welcome to Java ".trim() returns a new string, "Welcome to Java".

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

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

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

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

Splitting a String

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

displays

Java
HTML
Perl

Matching, Replacing and Splitting by Patterns

- The replaceAll, replaceFirst, split and matches methods can be used with a ***regular expression (a sequence of characters that specifies a search pattern)***
  - 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);
```

Displays: aNNNbNNNNNNc

- The regular expression [$+#] specifies a pattern that matches $, +, or #

Matching, Replacing and Splitting by Patterns

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.

Matching, Replacing and Splitting by Patterns

- The **`matches`** method tells whether or not a string matches a given regular expression.

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

```
true
```

Social security numbers is **xxx-xx-xxxx, where x is a digit:**

```
[\\d]{3}-[\\d]{2}-[\\d]{4}
```

An even number ends with digits **0, 2, 4, 6, or 8:**

```
[\\d]*[02468]
```

Telephone numbers **(xxx) xxx-xxxx, where x is a digit and the first digit cannot be zero:**

```
\\([1-9][\\d]{2}\\) [\\d]{3}-[\\d]{4}
```

Regular Expressions

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 [1-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java matches "[\\w]ava"
\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*	Java matches "[\\w]*"
p+	one or more occurrences of pattern *p*	Java matches "[\\w]+"
p?	zero or one occurrence of pattern *p*	Java matches "[\\w]?Java" Java matches "[\\w]?ava"
p{n}	exactly n occurrences of pattern *p*	Java matches "[\\w]{4}"
p{n,}	at least n occurrences of pattern *p*	Java matches "[\\w]{3,}"
p{n,m}	between n and m occurrences (inclusive)	Java matches "[\\w]{1,9}"

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]);
```

Displays:

```
Java
C
C#
C++
```

Examples

```
String s = "Java Java Java".replaceAll("v\\w", "wi") ;
      // "Jawi Jawi Jawi"

String s2 = "Java Java Java".replaceFirst("v\\w", "wi") ;
      // "Jawi Java Java"

String[] s3 = "Java1HTML2Perl".split("\\d");
      // ["Java", "HTML", "Perl"]
```

Command-Line Parameters

```
class TestMain {
   public static void main(String[] args){
       ...
   }
}
```

- We can run the program with:

```
java TestMain arg0 arg1 arg2 ... argn
```

or the EclipseIDE Run Configuration arguments

- In the main method, get the arguments `args[0], args[1], ..., args[n]` corresponds to `arg0, arg1, ..., argn` in the command line or the EclipseIDE Run Configuration arguments

Processing Command-Line Parameters

```
public class Calculator {
  public static void main(String[] args) {
      if (args.length != 3) {
          System.out.println("Usage: java Calculator
              operand1 operator operand2");
          System.exit(0);
      }
      int result = 0;
      switch (args[1].charAt(0)) {
      case '+':    result = Integer.parseInt(args[0])+
                       Integer.parseInt(args[2]);
                   break;
      ...
  }
}
```

```
javac Calculator.java
java Calculator 1 + 2
3
```

StringBuilder and StringBuffer

- The **StringBuilder/StringBuffer** classes are alternatives to the **String** class:
  - **StringBuilder/StringBuffer** can be used wherever a string is used
    - **StringBuffer** is synchronized i.e. thread safe. It means two threads can't call the methods of **StringBuffer** simultaneously. **StringBuilder** is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
  - **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 StringBuilder(s)

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.

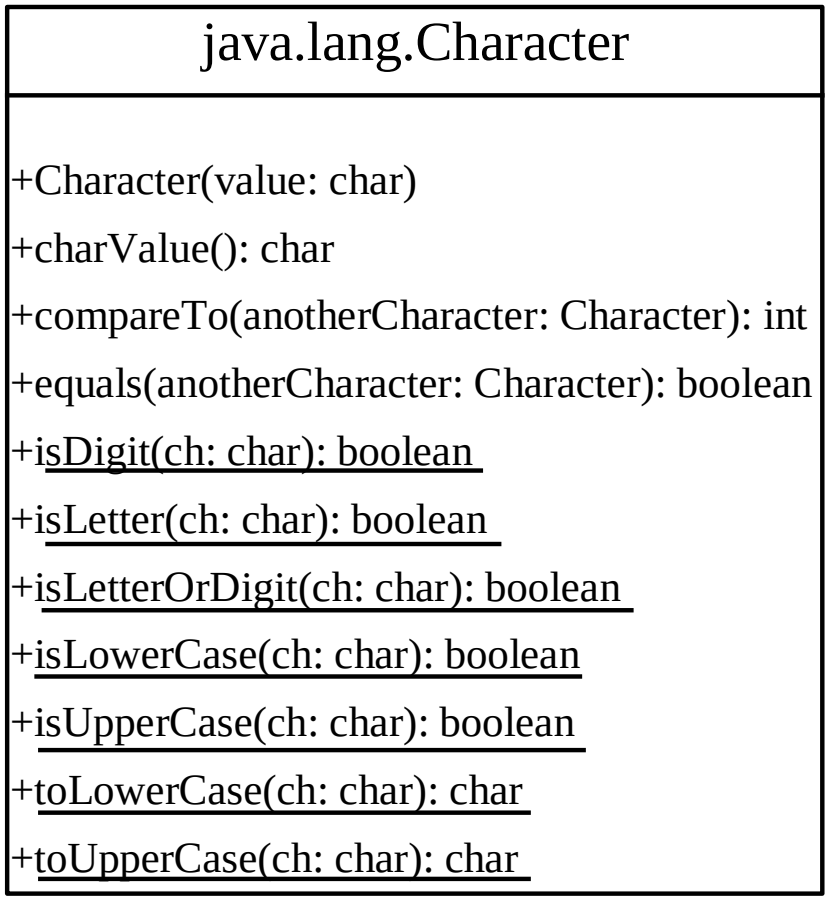
Modifying StringBuilder(s)

```
                                                    StringBuilder:
StringBuilder stringBuilder =
                        new StringBuilder();
stringBuilder.append("Java");                       Java
stringBuilder.insert(2,"HTML and ");                JaHTML and va
stringBuilder.delete(3, 4);                         JaHML and va
stringBuilder.deleteCharAt(5);                      JaHMLand va
stringBuilder.reverse();                            av dnaLMHaJ
stringBuilder.replace(4, 8, "HTML");                av dHTMLHaJ
stringBuilder.setCharAt(0, 'w');                    wv dHTMLHaJ
```

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.

The `Character` Class



Constructs a character object with char value

Returns the char value from this object

Compares this character with another

Returns true if this character equals to another

Returns true if the specified character is a digit

Returns true if the specified character is a letter

Returns true if the character is a letter or a digit

Returns true if the character is a lowercase letter

Returns true if the character is an uppercase letter

Returns the lowercase of the specified character

Returns the uppercase of the specified character

equals and compareTo

```
Character charObject = new Character('b');


charObject.equals(new Character('b'))
     returns true
charObject.equals(new Character('d'))
     returns false


charObject.compareTo(new Character('a'))
     returns 1
charObject.compareTo(new Character('b'))
     returns 0
charObject.compareTo(new Character('c'))
     returns -1
charObject.compareTo(new Character('d')
     returns -2
```

Designing Classes

- Coherence: A class should describe a single entity
  - Separating responsibilities: A single entity with too many responsibilities can be broken into several classes to separate responsibilities
- Reuse: Classes are designed for reuse!

Designing Classes

- Follow standard Java programming style and naming conventions:
  - Choose informative names for classes, data fields, and methods
- Place the data declaration before the constructor, and place constructors before methods.
- Provide a public no-arg constructor and override the **`equals`** method and the **`toString`** method (returns a **`String`**) whenever possible