Thinking in Objects

CSE160, Computer Science A: Honors
Stony Brook University

http://www.cs.stonybrook.edu/~cse160
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

```java
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

- A class with all `private` data fields and without mutators is not necessarily immutable!
What Class is Immutable?

1. It must mark all data fields private!
2. Provide no mutator 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, 1998, 1, 1);
        BirthDate date = student.getBirthDate();
        date.setYear(2050);
        // Now the student birth year is changed:
        System.out.println(student.getBirthDate().getYear()); // 2050
    }
}
Scope of Variables

- **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.

Versus

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

```java
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
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;
    }
}

Every instance variable belongs to an instance represented by this, which is normally omitted.
Class Abstraction and Encapsulation

**Abstraction** = 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 does not need to know how the class is implemented: it is encapsulated and hidden.

Class implementation is like a black box hidden from the clients

| Class Contract (Signatures of public methods and public constants) | Clients use the class through the contract of the class |
| Class | |

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
## Designing the Loan Class

<table>
<thead>
<tr>
<th>Loan</th>
</tr>
</thead>
<tbody>
<tr>
<td>-annualInterestRate: double</td>
</tr>
<tr>
<td>-numberOfYears: int</td>
</tr>
<tr>
<td>-loanAmount: double</td>
</tr>
<tr>
<td>-loanDate: Date</td>
</tr>
<tr>
<td>+Loan()</td>
</tr>
<tr>
<td>+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)</td>
</tr>
<tr>
<td>+getAnnualInterestRate(): double</td>
</tr>
<tr>
<td>+getNumberOfYears(): int</td>
</tr>
<tr>
<td>+getLoanAmount(): double</td>
</tr>
<tr>
<td>+getLoanDate(): Date</td>
</tr>
<tr>
<td>+setAnnualInterestRate(annualInterestRate: double): void</td>
</tr>
<tr>
<td>+setNumberOfYears(numberOfYears: int): void</td>
</tr>
<tr>
<td>+setLoanAmount(loanAmount: double): void</td>
</tr>
<tr>
<td>+getMonthlyPayment(): double</td>
</tr>
<tr>
<td>+getTotalPayment(): double</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>BMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name: String</td>
</tr>
<tr>
<td>-age: int</td>
</tr>
<tr>
<td>-weight: double</td>
</tr>
<tr>
<td>-height: double</td>
</tr>
</tbody>
</table>

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

+getBMI(): double
+getStatus(): String

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

<table>
<thead>
<tr>
<th>Course</th>
</tr>
</thead>
</table>
| -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.  

<table>
<thead>
<tr>
<th>Course</th>
</tr>
</thead>
</table>
| -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) {
        students[numberOfStudents++] = student;
    }
    public String[] getStudents() {
        return students;
    }
    public int getNumberOfStudents() {
        return numberOfStudents;
    }
    public String getCourseName() {
        return courseName;
    }
}
Example: The **StackOfIntegers** Class

<table>
<thead>
<tr>
<th>StackOfIntegers</th>
</tr>
</thead>
<tbody>
<tr>
<td>-elements: int[]</td>
</tr>
<tr>
<td>-size: int</td>
</tr>
<tr>
<td>+StackOfIntegers()</td>
</tr>
<tr>
<td>+StackOfIntegers(capacity: int)</td>
</tr>
<tr>
<td>+empty(): boolean</td>
</tr>
<tr>
<td>+peek(): int</td>
</tr>
<tr>
<td>+push(value: int): int</td>
</tr>
<tr>
<td>+pop(): int</td>
</tr>
<tr>
<td>+getSize(): int</td>
</tr>
</tbody>
</table>

- 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

```
Data1
Data2
Data1
Data2
Data3
Data1
Data2
Data3
Data1
Data2
```

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
Implementing **StackOfIntegers** Class

- `elements[capacity - 1]`
- `elements[size - 1]`
- `elements[1]`
- `elements[0]`
- `top`
- `bottom`
- `capacity`
- `size`
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;
    }
}
Constructing Strings

- Pattern:
  
  ```java
  String newString = new String(stringLiteral);
  ```

- Example:
  
  ```java
  String message = new String("Welcome to Java");
  ```

- Since strings are used frequently, Java provides a shorthand initializer for creating a string:
  
  ```java
  String message = "Welcome to Java";
  ```
Strings Are Immutable

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

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

After executing `String s = "Java";`

![Diagram showing an immutable String object with contents "Java" and cannot be changed]

After executing `s = "HTML";`

![Diagram showing the same String object now unreferenced after assignment]

This string object is now unreferenced
Interned Strings

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));

- 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.

display

s1 == s2 is false
s1 == s3 is true
The String Class methods

- Compare strings
- Obtaining String length
- 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
String Comparisons

- `equals(Object object):

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

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

  if (s1 == s2) {
    // s1 and s2 have the same reference
  }
String Comparisons

- `compareTo(Object object)`: 

```java
String s1 = new String("Welcome");
String s2 = "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
}
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+equals(s1: String): boolean</code></td>
<td>Returns true if this string is equal to string <code>s1</code>.</td>
</tr>
<tr>
<td><code>+equalsIgnoreCase(s1: String): boolean</code></td>
<td>Returns true if this string is equal to string <code>s1</code> case-insensitive.</td>
</tr>
<tr>
<td><code>+compareTo(s1: String): int</code></td>
<td>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 <code>s1</code>.</td>
</tr>
<tr>
<td><code>+compareToIgnoreCase(s1: String): int</code></td>
<td>Same as <code>compareTo</code> except that the comparison is case-insensitive.</td>
</tr>
<tr>
<td><code>+regionMatches(toffset: int, s1: String, offset: int, len: int): boolean</code></td>
<td>Returns true if the specified subregion of this string exactly matches the specified subregion in string <code>s1</code>.</td>
</tr>
<tr>
<td><code>+regionMatches(IGNORE_CASE: boolean, toffset: int, s1: String, offset: int, len: int): boolean</code></td>
<td>Same as the preceding method except that you can specify whether the match is case-sensitive.</td>
</tr>
<tr>
<td><code>+startsWith(prefix: String): boolean</code></td>
<td>Returns true if this string starts with the specified prefix.</td>
</tr>
<tr>
<td><code>+endsWith(suffix: String): boolean</code></td>
<td>Returns true if this string ends with the specified suffix.</td>
</tr>
</tbody>
</table>
String Length, Characters, and Combining Strings

<table>
<thead>
<tr>
<th>java.lang.String</th>
<th>____________________________________________________________________________</th>
</tr>
</thead>
<tbody>
<tr>
<td>+length(): int</td>
<td>Returns the number of characters in this string.</td>
</tr>
<tr>
<td>+charAt(index: int): char</td>
<td>Returns the character at the specified index from this string.</td>
</tr>
<tr>
<td>+concat(s1: String): String</td>
<td>Returns a new string that concatenate this string with string s1.</td>
</tr>
</tbody>
</table>
Finding String Length

- Finding string length using the `length()` method:

```javascript
message = "Welcome";
message.length() // returns 7
```
Retrieving Individual Characters in a String

- Use `message.charAt(index) \rightarrow char`
- Index starts from 0

Indices: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

message: Welcome to Java

message.charAt(0): W
message.charAt(14): a
message.length() is 15
message.charAt(14): a
Extracting Substrings

<table>
<thead>
<tr>
<th>java.lang.String</th>
</tr>
</thead>
<tbody>
<tr>
<td>+subString(beginIndex: int): String</td>
</tr>
<tr>
<td>+subString(beginIndex: int, endIndex: int): String</td>
</tr>
</tbody>
</table>

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";

message
Indices
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
message.substring(0, 11)
message.substring(11)

s2 will be "Welcome to HTML"
## Converting, Replacing, and Splitting Strings

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toLowerCase(): String</td>
<td>Returns a new string with all characters converted to lowercase.</td>
</tr>
<tr>
<td>toUpperCase(): String</td>
<td>Returns a new string with all characters converted to uppercase.</td>
</tr>
<tr>
<td>trim(): String</td>
<td>Returns a new string with blank characters trimmed on both sides.</td>
</tr>
<tr>
<td>replace(oldChar: char, newChar: char): String</td>
<td>Returns a new string that replaces all matching character in this string with the new character.</td>
</tr>
<tr>
<td>replaceFirst(oldString: String, newString: String): String</td>
<td>Returns a new string that replaces the first matching substring in this string with the new substring.</td>
</tr>
<tr>
<td>replaceAll(oldString: String, newString: String): String</td>
<td>Returns a new string that replace all matching substrings in this string with the new substring.</td>
</tr>
<tr>
<td>split(delimiter: String): String[]</td>
<td>Returns an array of strings consisting of the substrings split by the delimiter.</td>
</tr>
</tbody>
</table>
Examples

"Welcome".toLowerCase() returns a new string, "welcome".
"Welcome".toUpperCase() returns a new string, "WELCOME".
"Welcome".trim() returns a new string, "Welcome".
"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.print(tokens[i] + "     ");
displays

    Java    HTML    Perl
Matching, Replacing and Splitting by Patterns

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

```java
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
Result: aNNN\bNNNN\bNNNN\bNc```

- Here the regular expression `[$+#]` specifies a pattern that matches $, +, or #.
The following statement splits the string into an array of strings delimited by some punctuation marks:

```java
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");
for (int i = 0; i < tokens.length; i++)
    System.out.println(tokens[i]);
```

Output:
- Java
- C
- C#
- C++
You can match, replace, or split a string by specifying a pattern = regular expressions

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

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\\\\\\) \d\d\d\) \d-\d\d\d\d
```
## Regular Expressions

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Matches</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a specified character x</td>
<td>Java matches Java</td>
</tr>
<tr>
<td>.</td>
<td>any single character</td>
<td>Java matches J..a</td>
</tr>
<tr>
<td>(ab</td>
<td>cd)</td>
<td>ab or cd</td>
</tr>
<tr>
<td>[abc]</td>
<td>a, b, or c</td>
<td>Java matches Ja[uvw]a</td>
</tr>
<tr>
<td>[^abc]</td>
<td>any character except a, b, or c</td>
<td>Java matches Ja[^ars]a</td>
</tr>
<tr>
<td>[a-z]</td>
<td>a through z</td>
<td>Java matches [A-M]av[a-d]</td>
</tr>
<tr>
<td>[^a-z]</td>
<td>a through z</td>
<td>Java matches Jav[^b-d]</td>
</tr>
<tr>
<td>[a-e[m-p]]</td>
<td>a through e or m through p</td>
<td>[A-G[I-M]]av[a-d]</td>
</tr>
<tr>
<td>[a-e&amp;&amp;[c-p]]</td>
<td>intersection of a-e with c-p</td>
<td>[A-P&amp;&amp;[I-M]]av[a-d]</td>
</tr>
<tr>
<td>\d</td>
<td>a digit, same as [1-9]</td>
<td>Java2 matches &quot;Java[\d]&quot;</td>
</tr>
<tr>
<td>\D</td>
<td>a non-digit</td>
<td>$Java matches &quot;[\D][\D]ava&quot;</td>
</tr>
<tr>
<td>\w</td>
<td>a word character</td>
<td>Java matches &quot;[\w]ava&quot;</td>
</tr>
<tr>
<td>\W</td>
<td>a non-word character</td>
<td>$Java matches &quot;[\W][\w]ava&quot;</td>
</tr>
<tr>
<td>\s</td>
<td>a whitespace character</td>
<td>&quot;Java 2&quot; matches &quot;Java\s2&quot;</td>
</tr>
<tr>
<td>\S</td>
<td>a non-whitespace character</td>
<td>Java matches &quot;[\S]ava&quot;</td>
</tr>
<tr>
<td>p*</td>
<td>zero or more occurrences of pattern p</td>
<td>Java matches &quot;[\w]*&quot;</td>
</tr>
<tr>
<td>p+</td>
<td>one or more occurrences of pattern p</td>
<td>Java matches &quot;[\w]+&quot;</td>
</tr>
<tr>
<td>p?</td>
<td>zero or one occurrence of pattern p</td>
<td>Java matches &quot;[\w]?Java&quot;</td>
</tr>
<tr>
<td>p{n}</td>
<td>exactly n occurrences of pattern p</td>
<td>Java matches &quot;[\w]{4}&quot;</td>
</tr>
<tr>
<td>p{n,}</td>
<td>at least n occurrences of pattern p</td>
<td>Java matches &quot;[\w]{3,}&quot;</td>
</tr>
<tr>
<td>p{n,m}</td>
<td>between n and m occurrences (inclusive)</td>
<td>Java matches &quot;[\w]{1,9}&quot;</td>
</tr>
</tbody>
</table>
## Replacing and Splitting Strings

<table>
<thead>
<tr>
<th>java.lang.String</th>
</tr>
</thead>
<tbody>
<tr>
<td>+matches(regex: String): boolean</td>
</tr>
<tr>
<td>+replaceAll(regex: String, replacement: String): String</td>
</tr>
<tr>
<td>+replaceFirst(regex: String, replacement: String): String</td>
</tr>
<tr>
<td>+split(regex: String): String[]</td>
</tr>
</tbody>
</table>
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”]
Finding a Character or a Substring in a String

<table>
<thead>
<tr>
<th>java.lang.String</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+indexOf(ch: char): int</td>
<td>Returns the index of the first occurrence of ch in the string.</td>
</tr>
<tr>
<td></td>
<td>Returns -1 if not matched.</td>
</tr>
<tr>
<td>+indexOf(ch: char, fromIndex: int): int</td>
<td>Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.</td>
</tr>
<tr>
<td>+indexOf(s: String): int</td>
<td>Returns the index of the first occurrence of string s in this string.</td>
</tr>
<tr>
<td></td>
<td>Returns -1 if not matched.</td>
</tr>
<tr>
<td>+indexOf(s: String, fromIndex: int): int</td>
<td>Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.</td>
</tr>
<tr>
<td>+lastIndexOf(ch: int): int</td>
<td>Returns the index of the last occurrence of ch in the string.</td>
</tr>
<tr>
<td></td>
<td>Returns -1 if not matched.</td>
</tr>
<tr>
<td>+lastIndexOf(ch: int, fromIndex: int): int</td>
<td>Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.</td>
</tr>
<tr>
<td>+lastIndexOf(s: String): int</td>
<td>Returns the index of the last occurrence of string s. Returns -1 if not matched.</td>
</tr>
<tr>
<td>+lastIndexOf(s: String, fromIndex: int): int</td>
<td>Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>W</td>
<td>e</td>
<td>l</td>
<td>c</td>
<td>o</td>
<td>m</td>
<td>e</td>
<td>t</td>
<td>o</td>
<td>J</td>
<td>a</td>
<td>v</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>
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.
  - `valueOf` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`.
  - Example: convert a double value to a string:
    \[
    \text{String.valueOf}(5.44) \rightarrow \text{"5.44"}
    \]
Command-Line Parameters

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

- Run with:
  java TestMain arg0 arg1 arg2 ... argn
  or EclipseIDE arguments
Processing Command-Line Parameters

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

```
javac Calculator.java
java Calculator 1 + 2
3
```
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;
            ...
        }
    }
}
The Character Class

```
java.lang.Character

+Character(value: char)
+charValue(): char
+compareTo(anotherCharacter: Character): int
+equals(anotherCharacter: Character): boolean
+isDigit(ch: char): boolean
+isLetter(ch: char): boolean
+isLetterOrDigit(ch: char): boolean
+isLowerCase(ch: char): boolean
+isUpperCase(ch: char): boolean
+toLowerCase(ch: char): char
+toUpperCase(ch: char): char

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
```
Examples

Character charObject = new Character('b');

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

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

charObject.equals(new Character('d'))
returns false
StringBuilder and StringBuffer

- The **StringBuilder/StringBuffer** class is an alternative to the **String** class:
  - **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

<table>
<thead>
<tr>
<th>java.lang.StringBuilder</th>
<th>Constructs an empty string builder with capacity 16.</th>
</tr>
</thead>
<tbody>
<tr>
<td>+StringBuilder()</td>
<td>Constructs a string builder with the specified capacity.</td>
</tr>
<tr>
<td>+StringBuilder(capacity: int)</td>
<td>Constructs a string builder with the specified capacity.</td>
</tr>
<tr>
<td>+StringBuilder(s: String)</td>
<td>Constructs a string builder with the specified string.</td>
</tr>
</tbody>
</table>
### Modifying Strings in the Builder

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+append(data: char[]): StringBuilder</code></td>
<td>Appends a char array into this string builder.</td>
</tr>
<tr>
<td><code>+append(data: char[], offset: int, len: int): StringBuilder</code></td>
<td>Appends a subarray in data into this string builder.</td>
</tr>
<tr>
<td><code>+append(v: aPrimitiveType): StringBuilder</code></td>
<td>Appends a primitive type value as a string to this builder.</td>
</tr>
<tr>
<td><code>+append(s: String): StringBuilder</code></td>
<td>Appends a string to this string builder.</td>
</tr>
<tr>
<td><code>+delete(startIndex: int, endIndex: int): StringBuilder</code></td>
<td>Deletes characters from startIndex to endIndex.</td>
</tr>
<tr>
<td><code>+deleteCharAt(index: int): StringBuilder</code></td>
<td>Deletes a character at the specified index.</td>
</tr>
<tr>
<td><code>+insert(index: int, data: char[], offset: int, len: int): StringBuilder</code></td>
<td>Inserts a subarray of the data in the array to the builder at the specified index.</td>
</tr>
<tr>
<td><code>+insert(offset: int, data: char[]): StringBuilder</code></td>
<td>Inserts data into this builder at the position offset.</td>
</tr>
<tr>
<td><code>+insert(offset: int, b: aPrimitiveType): StringBuilder</code></td>
<td>Inserts a value converted to a string into this builder.</td>
</tr>
<tr>
<td><code>+insert(offset: int, s: String): StringBuilder</code></td>
<td>Inserts a string into this builder at the position offset.</td>
</tr>
<tr>
<td><code>+replace(startIndex: int, endIndex: int, s: String): StringBuilder</code></td>
<td>Replaces the characters in this builder from startIndex to endIndex with the specified string.</td>
</tr>
<tr>
<td><code>+reverse(): StringBuilder</code></td>
<td>Reverses the characters in the builder.</td>
</tr>
<tr>
<td><code>+setCharAt(index: int, ch: char): void</code></td>
<td>Sets a new character at the specified index in this builder.</td>
</tr>
</tbody>
</table>
Examples

StringBuilder stringBuilder =
    new StringBuilder();
stringBuilder.append("Java");
stringBuilder.insert(2,"HTML and ");
stringBuilder.delete(3, 4);
stringBuilder.deleteCharAt(5);
stringBuilder.reverse();
stringBuilder.replace(4, 8, "HTML");
stringBuilder.setCharAt(0, 'w');
The **toString, capacity, length, setLength, and charAt** Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.StringBuilder</code></td>
<td></td>
</tr>
<tr>
<td><code>+toString()</code>: String</td>
<td>Returns a string object from the string builder.</td>
</tr>
<tr>
<td><code>+capacity()</code>: int</td>
<td>Returns the capacity of this string builder.</td>
</tr>
<tr>
<td><code>+charAt(index: int)</code>: char</td>
<td>Returns the character at the specified index.</td>
</tr>
<tr>
<td><code>+length()</code>: int</td>
<td>Returns the number of characters in this builder.</td>
</tr>
<tr>
<td><code>+setLength(newLength: int)</code>: void</td>
<td>Sets a new length in this builder.</td>
</tr>
<tr>
<td><code>+substring(startIndex: int)</code>: String</td>
<td>Returns a substring starting at startIndex.</td>
</tr>
<tr>
<td><code>+substring(startIndex: int, endIndex: int)</code>: String</td>
<td>Returns a substring from startIndex to endIndex-1.</td>
</tr>
<tr>
<td><code>+trimToSize()</code>: void</td>
<td>Reduces the storage size used for the string builder.</td>
</tr>
</tbody>
</table>
Designing a Class

- **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

- Classes are designed for reuse!

- Provide a public no-arg constructor and override the `equals` method and the `toString` method defined in the `Object` class whenever possible
Designing a Class

• 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,
  • Always provide a constructor and initialize variables to avoid programming errors.