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
- A class with all private data fields and without mutators is not necessarily 2 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;
}
}
```

Example mutable

```
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.

VS.

- 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

```
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 Constructor

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

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

```
 public Circle() {
 this(1.0);
 }
```

this is used to invoke another constructor

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

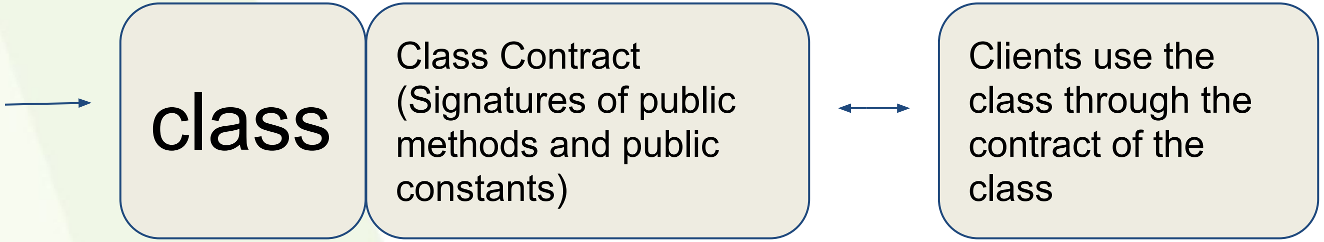
Every instance variable belongs to an instance represented by this, which is normally omitted

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

Designing the Loan Class

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

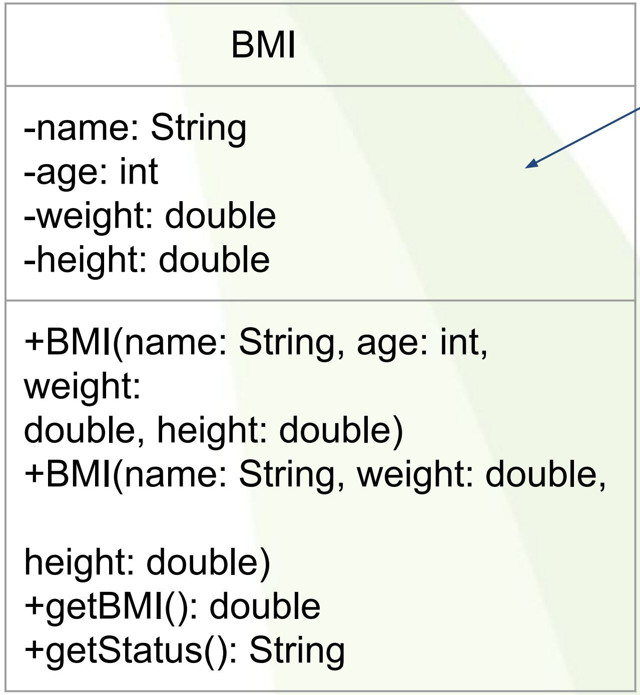
Designing the Loan Class ‘Cont’

```
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();
```

Designing the Loan Class

```
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



```
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 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.)

The BMI Class ‘Cont’

```
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;
```

The BMI Class

```
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
-students: String[]
-numberOfStudents: int
----------------------------------------
+Course(name: String)
+getName(): String
+addStudent(student: String): void
+getStudents(): String[]
+getNumberOfStudents(): int
```

The name of the course.
The students who take the course.
The number of students (default: 0).

Creates a Course with the specified name.
Returns the course name.
a new student to the course list.
Returns the students for the course.
Returns the number of students for the course.

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	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.

Example: The Course Class ‘Cont’

```
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;
numberOfStudents++;
}
```

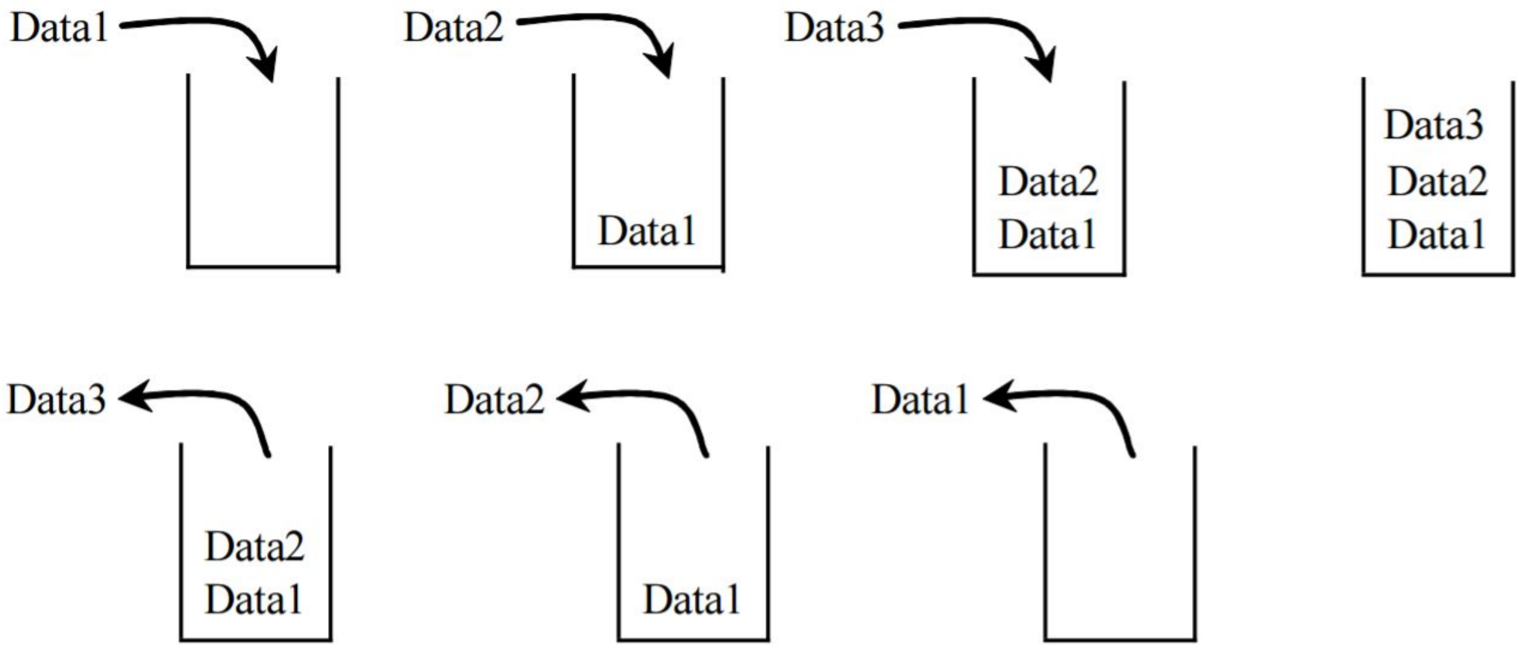
Example: The Course Class

```
}
public String[] getStudents() {
return students;
}
public int getNumberOfStudents() {
return numberOfStudents;
}
public String getCourseName() {
return courseName;
}
}
```

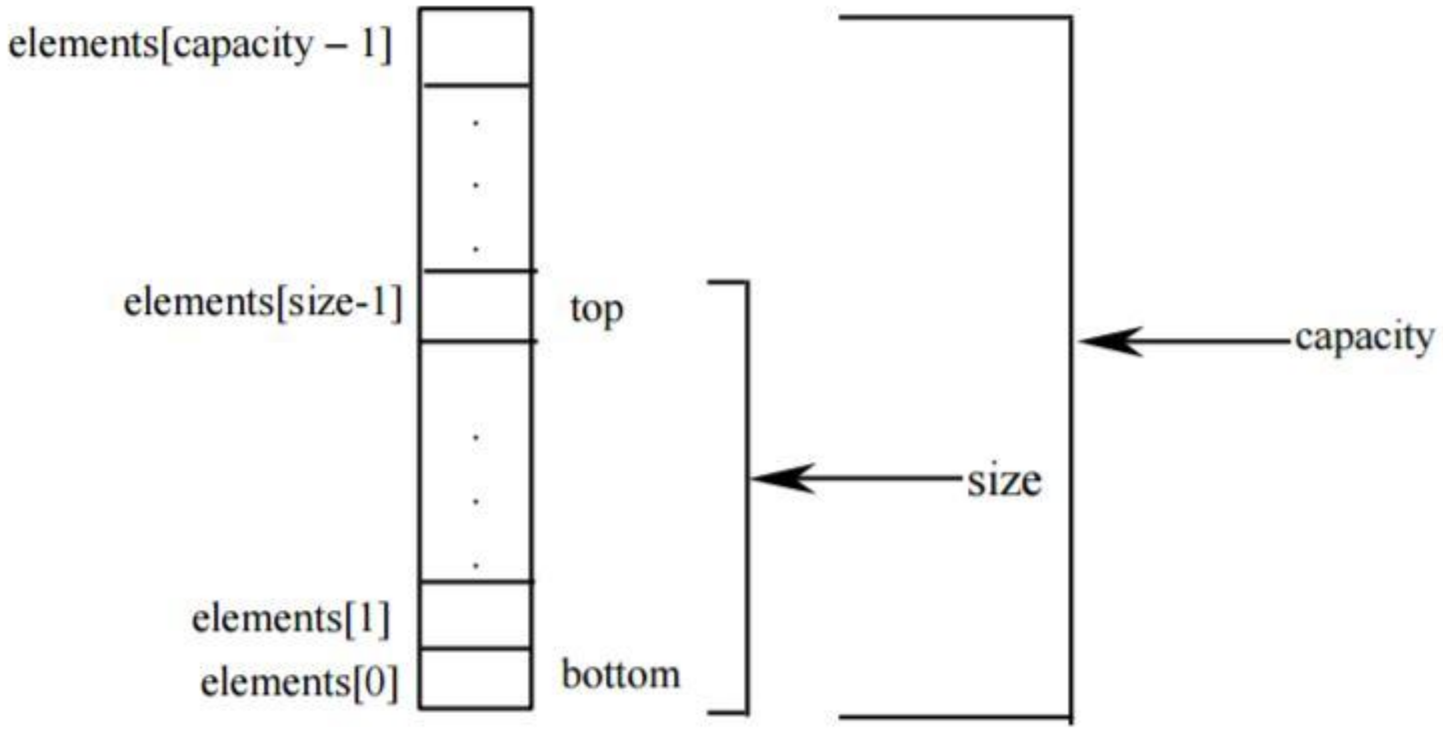
Example: The StackOfIntegers Class

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

Designing the StackOfIntegers Class



Implementing StackOfIntegers Class



Implementing StackOfIntegers Class ‘Cont’

```
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];
```

Implementing StackOfIntegers Class

```
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:

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

- Example:

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

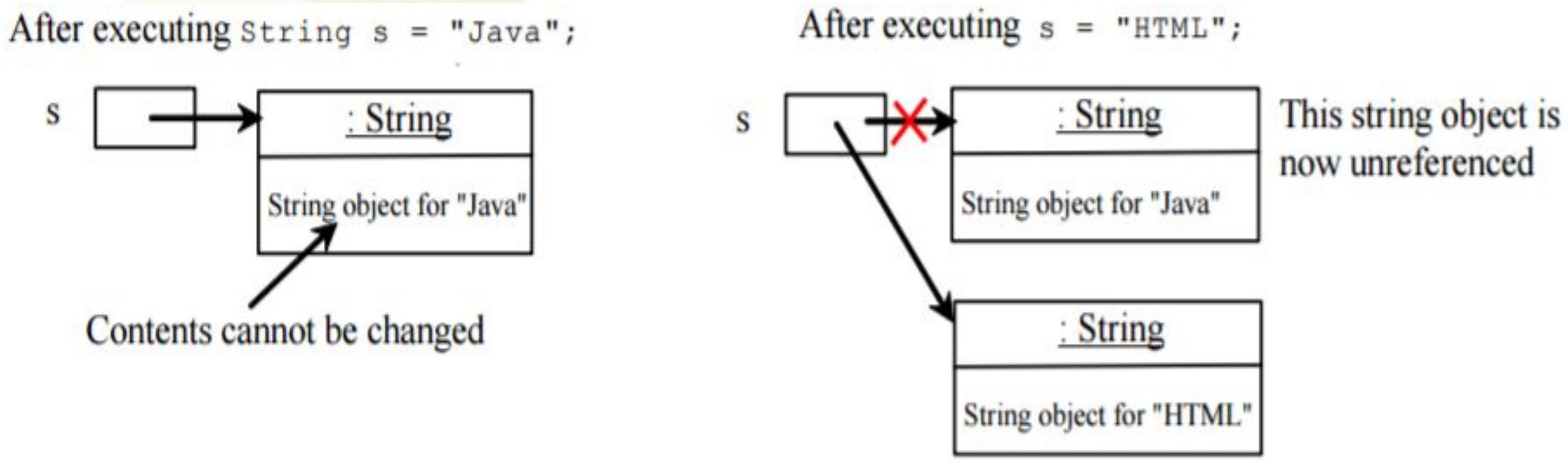
- Since strings are used frequently, Java provides a shorthand initializer for creating a string:

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

Strings Are Immutable

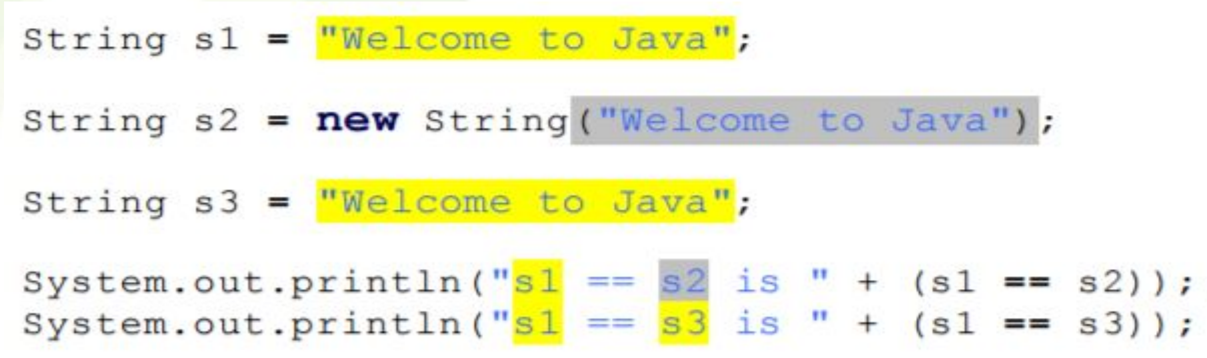
- A String object is immutable; its contents cannot be changed

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



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

Interned Strings

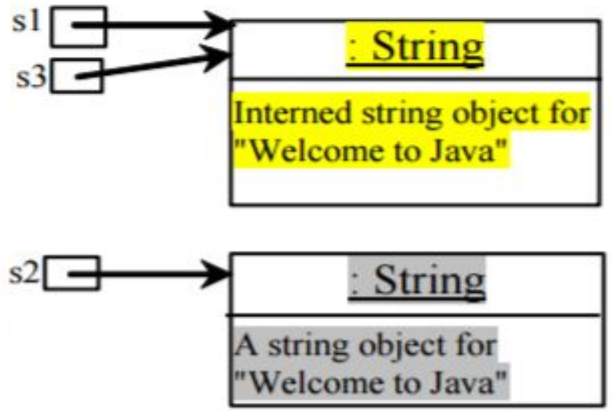


```
String s1 = "Welcome to Java";

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

String s3 = "Welcome to Java";

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



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

Display

s1 == s2 is false

s1 == s3 is true

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

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):

```
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
}
```

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 Caseinsensitive.
+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 caseinsensitive.
+regionMatches(toffset: int, s1: Strig, 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.