

# Object-Oriented Thinking and Introduction to Generics

Introduction to Programming and  
Computational Problem Solving II

CSE 8B

Lecture 11

# Announcements

- Midterm assessment is Feb 10-15
- Assignments 2-4 upgrades due Feb 19, 11:59 PM
- Assignment 5 will be released Feb 14
  - Due Feb 24, 11:59 PM

# Object-oriented thinking

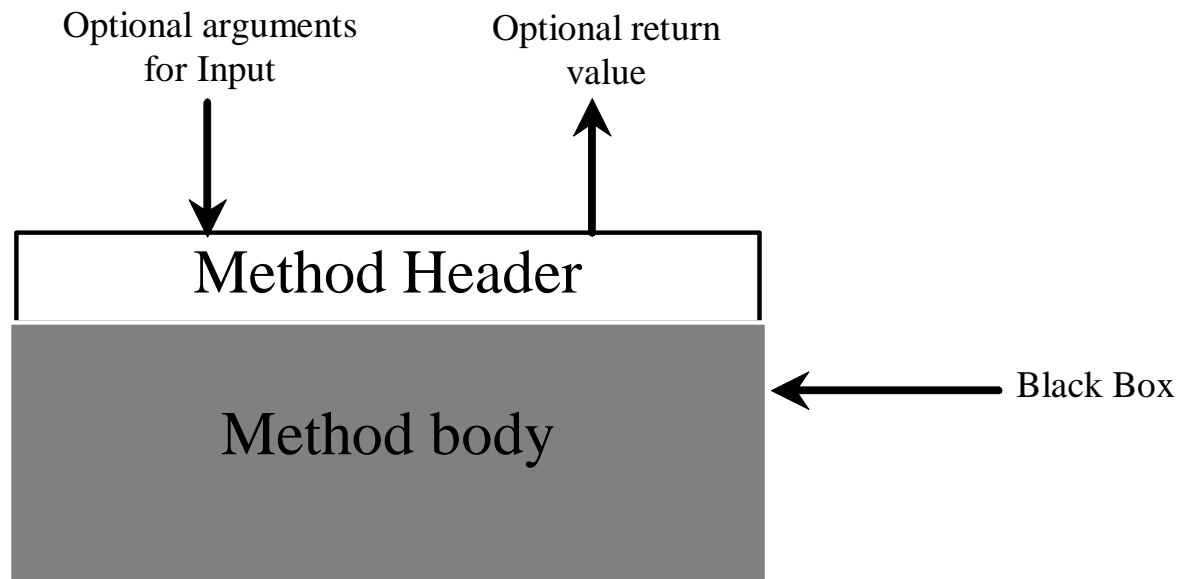
- The advantages of object-oriented programming over procedural programming
- Classes provide more flexibility and modularity for building reusable software
- How to solve problems using the object-oriented paradigm
- Class design

# Procedural programming vs object-oriented programming

- Procedural programming
  - Data and operations on data are separate
  - Requires passing data to methods
- Object-oriented programming
  - Data and operations on data are in an object
  - Organizes programs like the real world
    - All objects are associated with both attributes and activities
  - Using objects improves software reusability and makes programs easier to both develop and maintain

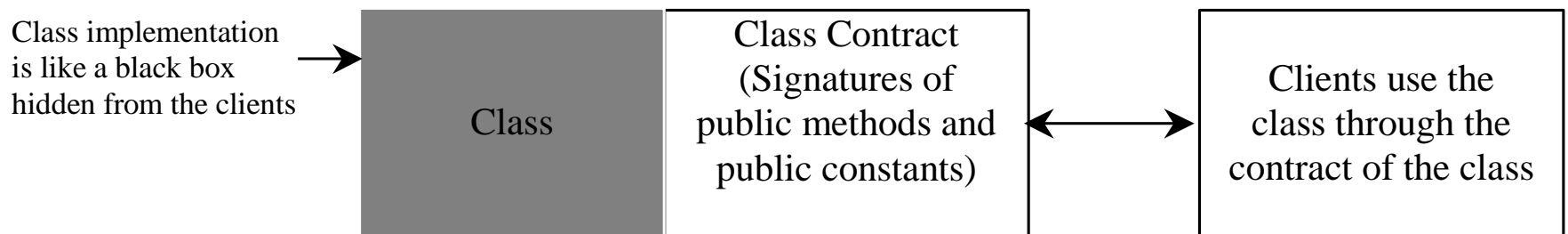
# Procedural programming: method abstraction

- You can think of the method body as a black box that contains the detailed implementation for the method



# Object-oriented programming: 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 lets the user know how the class can be used
  - The *class contract*
- 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 encapsulation*
  - A class is called an *abstract data type* (ADT)



# Class abstraction and encapsulation

- For example, a class for a loan

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.

The creator of the class provides a description of the class and lets the user know how the class can be used

The *class contract*

# Class abstraction and encapsulation

- A class is designed for use by many different users (or customers or clients)
- To be useful in a wide range of applications, a class should provide a variety of ways for customization through properties, and constructors and methods that, together, are **minimal and complete**



# Thinking in objects

- Procedural programming focuses on designing methods
- Object-oriented programming
  - Couples data and methods together into objects
  - Focuses on designing objects and operations on objects
- Object-orientated programming combines the power of procedural programming with an additional component that integrates data with operations into objects

# Class relationships

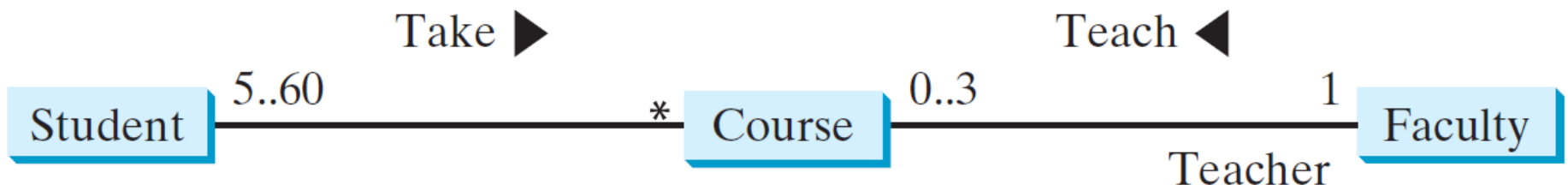
- To design classes, one must understand the relationships among classes
  - Association
  - Aggregation
  - Composition
  - Inheritance (covered next lecture)

# Association

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

# Association

- Multiplicity
  - The number of objects of a class
- For example
  - Each student may take any number (\*) of courses
  - Each course must have 5 to 60 students
  - Each course is taught by 1 faculty member
  - Each faculty member must teach 0 to 3 courses



# Association

- In Java, associations can be implemented using data fields and methods
  - For example
    - A student takes a course
      - addCourse method in Student class
      - addStudent method in Course class
    - A faculty member teaches a course
      - addCourse method in Faculty class
      - setFaculty method in Course class
    - The Student class may store the courses a student is taking
      - `private Course[] courseList;`
    - The Faculty class may store the courses a faculty member is teaching
      - `private Course[] courseList;`
- **There are many possible ways to implement association relationships**

# Aggregation

- Special form of association representing an owner-subject relationship
  - The *owner* object is called an *aggregating object* and its class is called an *aggregating class*
  - The *subject* object is called an *aggregated object* and its class is called an *aggregated class*
- Models **has-a** relationships
  - For example
    - A student **has-a** name
    - A student **has-an** address

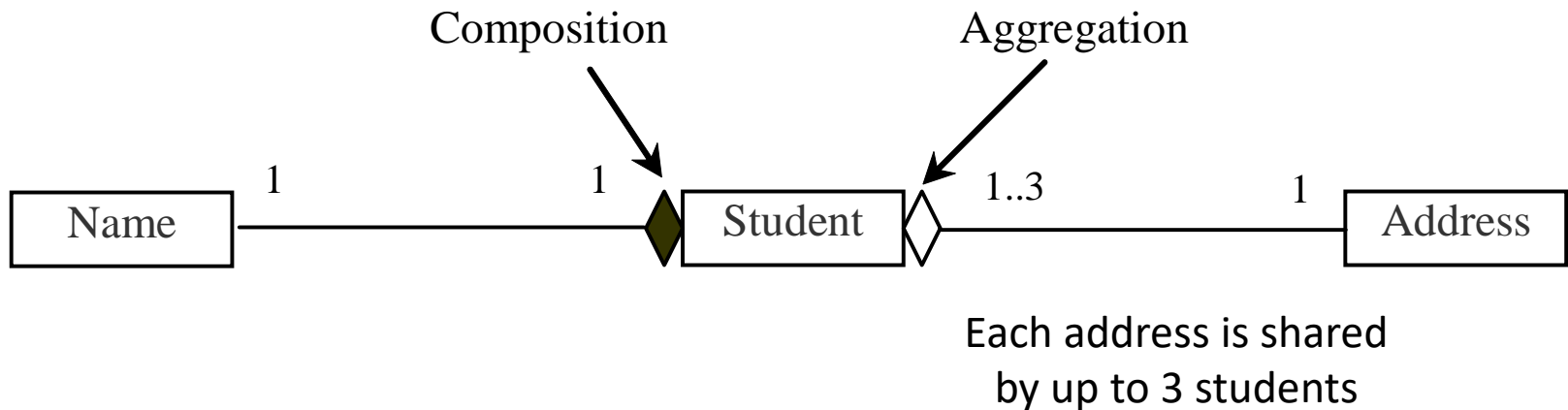
*Owner has-a subject*  
*Aggregating has-an aggregated*

# Composition

- Aggregation between two objects is called *composition* if the existence of the aggregated object is dependent on the aggregating object
  - **Exclusive** ownership of the subject
  - The subject (i.e., aggregated object) cannot (conceptually) exist on its own
    - For example
      - A book **has-a** page and when the book is destroyed, so is the page
      - A page has no meaning or purpose without the book

# Aggregation and composition

- For example
  - When the student object is destroyed
    - Their name is destroyed (composition)
    - Their address is not destroyed (aggregation)





# Aggregation and composition

- Usually represented as a data field in the aggregating class

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

Aggregated class

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

Aggregating class

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

Aggregated class

# Aggregation between same class

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

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

- For example, a person may have multiple supervisors

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

# Aggregation or composition

- Warning: Since aggregation and composition relationships are represented using classes in similar ways, many texts do not differentiate them, calling both compositions

# Class design and development

- For example, a class for a course

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

The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.

Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.

# Class design and development

```
public class TestCourse {
    public static void main(String[] args) {
        Course course1 = new Course("Data Structures");
        Course course2 = new Course("Database Systems");

        course1.addStudent("Peter Jones");
        course1.addStudent("Brian Smith");
        course1.addStudent("Anne Kennedy");

        course2.addStudent("Peter Jones");
        course2.addStudent("Steve Smith");

        System.out.println("Number of students in course1: "
            + course1.getNumberOfStudents());
        String[] students = course1.getStudents();
        for (int i = 0; i < course1.getNumberOfStudents(); i++)
            System.out.print(students[i] + ", ");

        System.out.println();
        System.out.print("Number of students in course2: "
            + course2.getNumberOfStudents());
    }
}
```

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

# Class design and development

```
public class Course {
    private String courseName;
    private String[] students = new String[4];
    private int numberOfStudents;

    public Course(String courseName) {
        this.courseName = courseName;
    }

    public void addStudent(String student) {
        students[numberOfStudents] = student;
        numberOfStudents++;
    }

    public String[] getStudents() {
        return students;
    }

    public int getNumberOfStudents() {
        return numberOfStudents;
    }

    public String getCourseName() {
        return courseName;
    }

    public void dropStudent(String student) {
        // TODO
    }
}
```

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

# Class design and development

- Use a UML class diagram to design the class
- Write a test program that uses the class
  - Developing a class and using a class are two separate tasks
  - It is easier to implement a class if you must use the class
- Implement the class
- Use Javadoc to document the class (contract)

# Object-oriented thinking

- Classes provide more flexibility and modularity for building reusable software
- Class abstraction and encapsulation
  - 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



# Primitive data type values as objects

- A primitive data type is not an object
- But it can be wrapped in an object using a Java API wrapper class

Boolean

Character (**not** Char)

Short

Byte

Integer (**not** Int)

Long

Float

Double

## Notes

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

# Integer and Double wrapper classes

## java.lang.Integer

-value: int

+MAX VALUE: int

+MIN VALUE: int

+Integer(value: int)

+Integer(s: String)

+byteValue(): byte

+shortValue(): short

+intValue(): int

+longVlaue(): long

+floatValue(): float

+doubleValue():double

+compareTo(o: Integer): int

+toString(): String

+valueOf(s: String): Integer

+valueOf(s: String, radix: int): Integer

+parseInt(s: String): int

+parseInt(s: String, radix: int): int

**Constructors are deprecated; use static valueOf methods instead**

## java.lang.Double

-value: double

+MAX VALUE: double

+MIN VALUE: double

+Double(value: double)

+Double(s: String)

+byteValue(): byte

+shortValue(): short

+intValue(): int

+longVlaue(): long

+floatValue(): float

+doubleValue():double

+compareTo(o: Double): int

+toString(): String

+valueOf(s: String): Double

+valueOf(s: String, radix: int): Double

+parseDouble(s: String): double

+parseDouble(s: String, radix: int): double

**Constructors are deprecated; use static valueOf methods instead**

# Wrapper classes

- Constructors are deprecated; use static `valueOf` methods instead
- Class Constants `MAX_VALUE` and `MIN_VALUE`
- Conversion Methods

Constructors are deprecated; use static `valueOf` methods instead

# Numeric wrapper class constructors

- You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value
  - For example, the constructors for `Integer` and `Double` are

```
public Integer(int value)
```

```
public Integer(String s)
```

```
public Double(double value)
```

```
public Double(String s)
```

# Numeric wrapper class constants

- Each numerical wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`
- `MAX_VALUE` represents the maximum value of the corresponding primitive data type
- For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum byte, short, int, and long values
- For `Float` and `Double`, `MIN_VALUE` represents the minimum **positive** float and double values

# Numeric wrapper class conversion methods

- Each numeric wrapper class implements the abstract methods `doubleValue`, `floatValue`, `intValue`, `longValue`, and `shortValue`
  - Defined in the abstract `Number` class (covered later in quarter)
- These methods “convert” objects into primitive type values

Constructors are deprecated; use static valueOf methods instead

# Numeric wrapper class static valueOf methods

- The numeric wrapper classes have a useful class methods valueOf
- These methods create a new object initialized to the value represented by the specified corresponding primitive data type value or string representing the numeric value

– For example

```
Double doubleObject = Double.valueOf(12.4);  
Integer integerObject = Integer.valueOf("12");
```

# Numeric wrapper class static parsing methods

- Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on 10 or any specified radix (e.g., 2 for binary, 8 for octal, 10 for decimal, 16 for hexadecimal)

– For example

`Integer.parseInt("13")` returns 13

`Integer.parseInt("13", 10)` returns 13

`Integer.parseInt("1A", 16)` returns 26



# Automatic conversion between primitive types and wrapper class types

- Converting a primitive value to a wrapper object is called *boxing*
- Converting a wrapper object to a primitive value is called *unboxing*
- The Java compiler will automatically convert a primitive data type value to an object using a wrapper class (*autoboxing*) and vice versa (*autounboxing*), depending on the context

# Automatic conversion between primitive types and wrapper class types

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

Equivalent

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

(a)

(b)

Autoboxing

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

Autounboxing

# BigInteger and BigDecimal classes

- If you need to compute with very large integers or high precision floating-point values, you can use the `BigInteger` and `BigDecimal` classes in the `java.math` package
- Both are **immutable**
- Both extend the `Number` class and implement the `Comparable` interface (covered later in quarter)

# BigInteger and BigDecimal classes

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

```
BigDecimal a = new BigDecimal(1.0);  
BigDecimal b = new BigDecimal(3);  
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);  
System.out.println(c);
```

↑  
Scale

# String class

- The `String` class has 13 constructors and more than 40 methods
- A good example for learning classes and objects

# Constructing strings

- Create from a string literal
  - Syntax

```
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";
```
- Create from an array of characters
  - Syntax

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

    - where, for example

```
char[] charArray = {'U', 'C', 'S', 'D', ' ', 'C', 'S', 'E'};
```

# Strings are immutable

- A `String` object is immutable (i.e., its contents cannot be changed once the string is created)
- The following code does not change the contents of the string

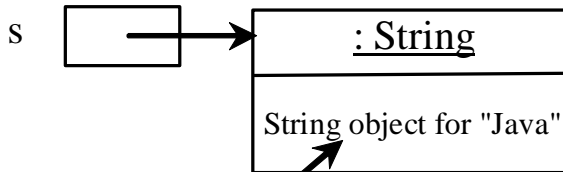
```
String s = "Java";  
s = "HTML";
```

# Strings are immutable

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

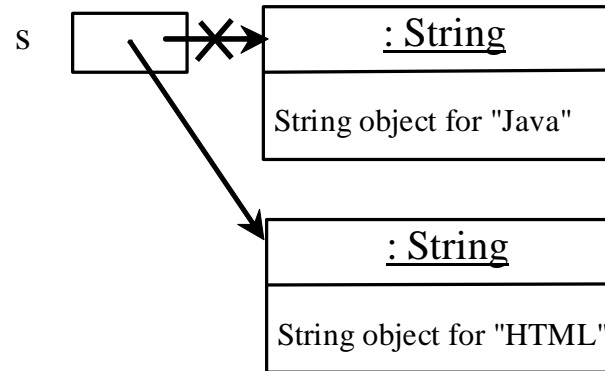
```
s = "HTML";
```

After executing `String s = "Java";`



Contents cannot be changed

After executing `s = "HTML";`



This string object is now unreferenced

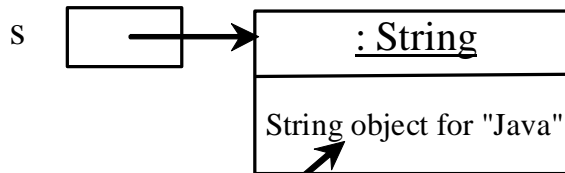


# Strings are immutable

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

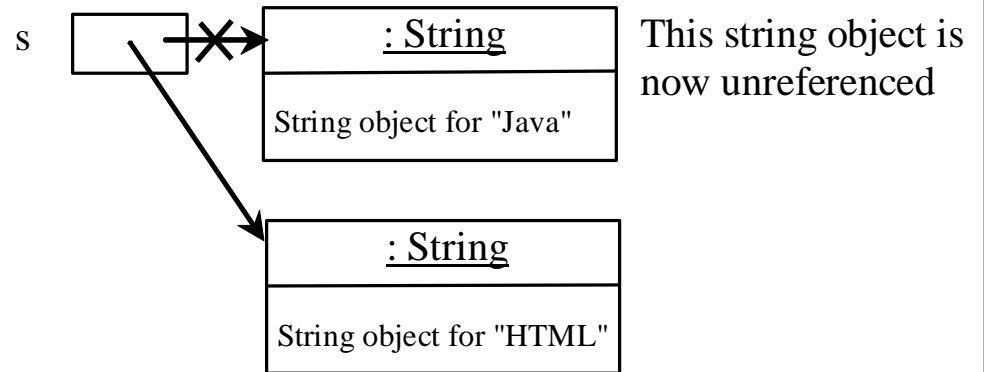
```
s = "HTML";
```

After executing `String s = "Java";`



Contents cannot be changed

After executing `s = "HTML";`



# Interned strings

- Since strings are immutable and are frequently used, to improve efficiency and save memory, the Java Virtual Machine (JVM) uses a unique instance for string literals with the same character sequence
- Such an instance is called *interned*

# Interned strings

- 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");
```

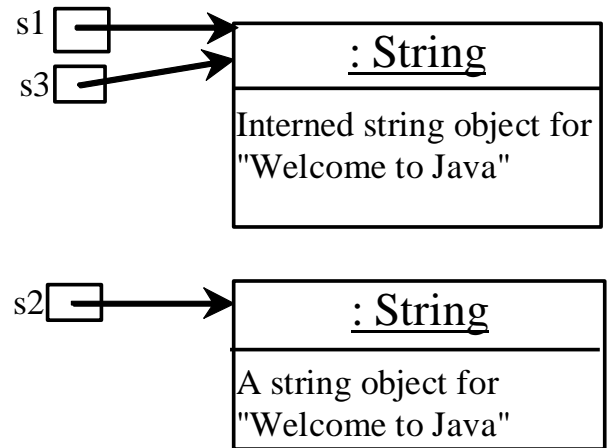
```
String s3 = "Welcome to Java";
```

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

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

s1 == s2 is false

s1 == s3 is true



# Replacing and splitting strings

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

# Replacing a string

- `"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("e1", "AB")`  
returns a new string `WABcome`

# Splitting a string

- Split a string into an array of strings

- For example, using # as a delimiter

```
String[] tokens = "CSE#8B#and#11#use#Java".split("#", 0);  
for (int i = 0; i < tokens.length; i++)  
    System.out.print(tokens[i] + " ");
```

- Displays CSE 8B and 11 use Java

# Matching, replacing, and splitting by patterns

- You can match, replace, or split a string by specifying a pattern
  - For example

```
"Java".equals("Java");  
"Java".matches("Java");
```
- This is an extremely useful and powerful feature known as *regular expression*
  - <https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#sum>
  - <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html#sum>

# Convert character and numbers to strings

- The `String` class provides several static `valueOf` methods for converting a character, an array of characters, and numeric values to strings
- These methods have the same name `valueOf` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`
  - For example, to convert a `double` value to a string, use `String.valueOf(5.44)`
    - The return value is string consists of characters `'5'`, `'.'`, `'4'`, and `'4'`
    - Compare with `String s = 5.44 + "";`



# StringBuilder and StringBuffer classes

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

# StringBuilder constructors

java.lang.StringBuilder

+StringBuilder()

Constructs an empty string builder with capacity 16.

+StringBuilder(capacity: int)

Constructs a string builder with the specified capacity.

+StringBuilder(s: String)

Constructs a string builder with the specified string.

# Modifying strings in the builder

java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: <i>aPrimitiveType</i> ): 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: <i>aPrimitiveType</i> ): 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.

# 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 ArrayList class

- You can create an array to store objects, but the array's size is fixed once the array is created
- Java provides the ArrayList class that can be used to store an unlimited number of objects

# Introduction to generics and the ArrayList class

## **java.util.ArrayList<E>**

```
+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int) : E  
+indexOf(o: Object) : int  
+isEmpty(): boolean  
+lastIndexOf(o: Object) : int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int) : boolean  
+set(index: int, o: E) : E
```

Creates an empty list.

Appends a new element *o* at the end of this list.

Adds a new element *o* at the specified index in this list.

Removes all the elements from this list.

Returns true if this list contains the element *o*.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns true if this list contains no elements.

Returns the index of the last matching element in this list.

Removes the element *o* from this list.

Returns the number of elements in this list.

Removes the element at the specified index.

Sets the element at the specified index.

# Introduction to generics and the `ArrayList` class

- Generics let you parameterize types
  - You can define a method or class with generic types, which are replaced with concrete types
- `ArrayList<E>` is known as a generic class with a generic type `E`
- You can specify a concrete type to replace `E` when creating an `ArrayList`
- For example
  - The below statements creates an `ArrayList` used to store strings and assigns its reference to variable `cities`

```
ArrayList<String> cities = new ArrayList<String>();  
ArrayList<String> cities = new ArrayList<>();
```

# Comparing arrays and ArrayList

<i>Operation</i>	<i>Array</i>	<i>ArrayList</i>
Creating an array/ArrayList	<code>String[] a = new String[10]</code>	<code>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</code>
Accessing an element	<code>a[index]</code>	<code>list.get(index);</code>
Updating an element	<code>a[index] = "London";</code>	<code>list.set(index, "London");</code>
Returning size	<code>a.length</code>	<code>list.size();</code>
Adding a new element		<code>list.add("London");</code>
Inserting a new element		<code>list.add(index, "London");</code>
Removing an element		<code>list.remove(index);</code>
Removing an element		<code>list.remove(Object);</code>
Removing all elements		<code>list.clear();</code>



# Array to/from ArrayList

- Creating an ArrayList from an array of objects

```
String[] array = {"red", "green", "blue"};  
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

- Creating an array of objects from an ArrayList

```
String[] array1 = new String[list.size()];  
list.toArray(array1);
```

# Useful methods in java.util.Collections

- Maximum element in ArrayList  
`java.util.Collections.max`
- Minimum element in ArrayList  
`java.util.Collections.min`
- Sort an ArrayList  
`java.util.Collections.sort`
- Shuffle an ArrayList  
`java.util.Collections.shuffle`

# Next Lecture

- Inheritance