

COMP30820
Java Programming (Conv)

Michael O'Mahony

Chapter 10 Thinking in Objects

Objectives

- To use the `String` class to process immutable strings.
- To use the `StringBuilder` classes to process mutable strings.
- To explore the differences between the procedural paradigm and object-oriented paradigm.
- To discover the relationships between classes.
- To understand class abstraction when developing software.

The `String` and `StringBuilder` Classes

Strings are widely used in programming and Java supports a number of classes and methods to handle and manipulate strings.

We begin by considering the `String` class, and then the `StringBuilder` class which supports mutable strings.

Constructing Strings

General syntax:

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

Example:

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

Since strings are often used, Java treats a string literal as a `String` object and provides the following shorthand for creating a string:

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

You can also create a string from an array of characters. For example, the following statements create the string "Hi All":

```
char[] arr = {'H', 'i', ' ', 'A', 'l', 'l'};  
String str = new String(arr);
```

Strings Are Immutable

A `String` object is immutable – its contents cannot be changed once the string is created.

Strings Are Immutable

A `String` object is immutable – its contents cannot be changed once the string is created.

Does the following code change the contents of the string?

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

Strings Are Immutable

A `String` object is immutable – its contents cannot be changed once the string is created.

Does the following code change the contents of the string?

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

The answer is no.

animation

Trace Code

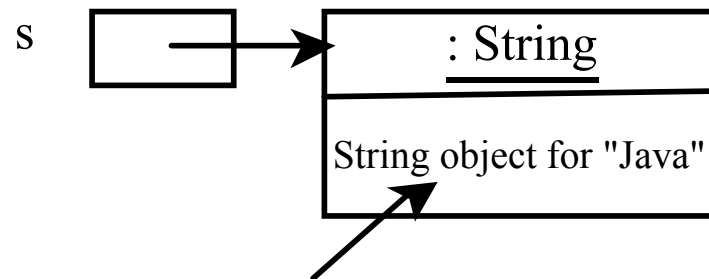
```
String s = "Java";  
s = "HTML";
```

Trace Code

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

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

After executing `String s = "Java";`



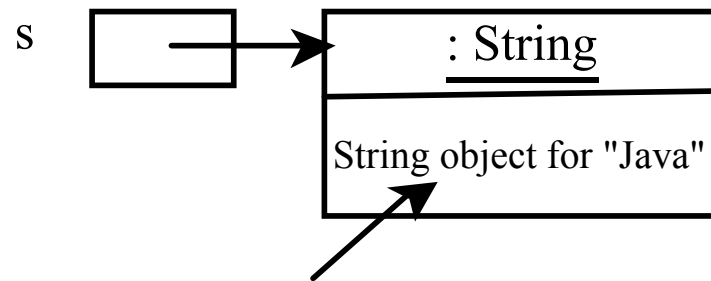
Contents cannot be changed

Trace Code

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

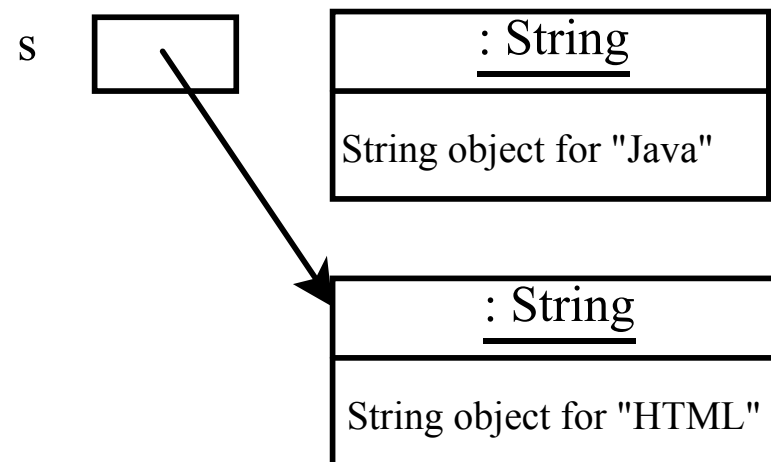
```
s = "HTML";
```

After executing `String s = "Java";`



Contents cannot be changed

After executing `s = "HTML";`



Interned Strings

Since strings are immutable and are frequently used, Java uses a *unique instance* for *string literals* with the *same character sequence*.

Interned Strings

Since strings are immutable and are frequently used, Java uses a *unique instance* for *string literals* with the *same character sequence*.

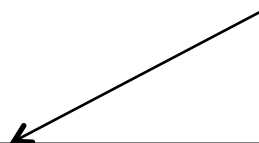
```
String message = "Welcome to Java";
```

Interned Strings

Since strings are immutable and are frequently used, Java uses a *unique instance* for *string literals* with the *same character sequence*.

String message = "Welcome to Java";

string literal



Interned Strings

Since strings are immutable and are frequently used, Java uses a *unique instance* for *string literals* with the *same character sequence*.

String message =  *string literal*

Such an instance is called an *interned string*.

The rationale for interned strings is to improve efficiency/
save memory.

Trace Code

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

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

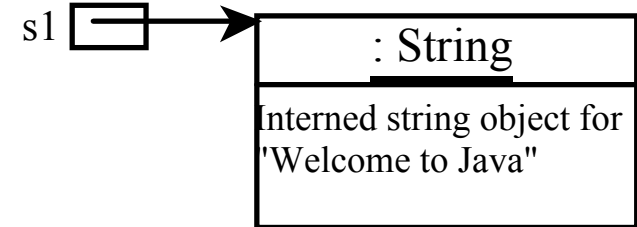
```
String s3 = "Welcome to Java";
```


Trace Code

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

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

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

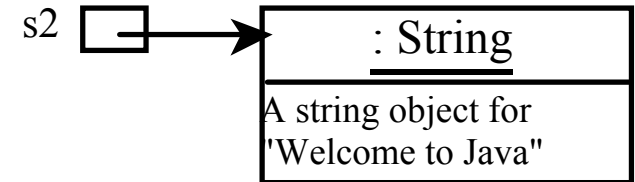
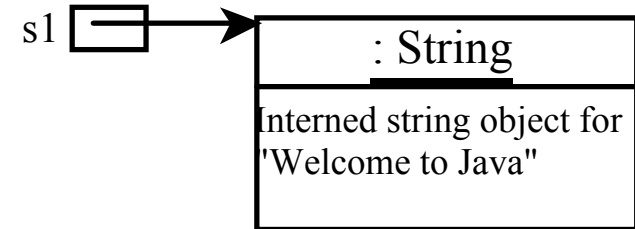


Trace Code

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

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

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

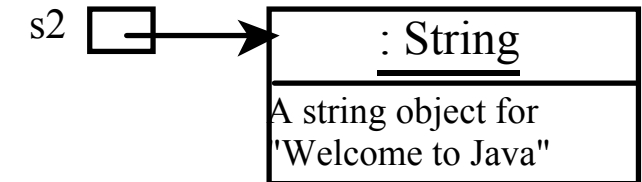
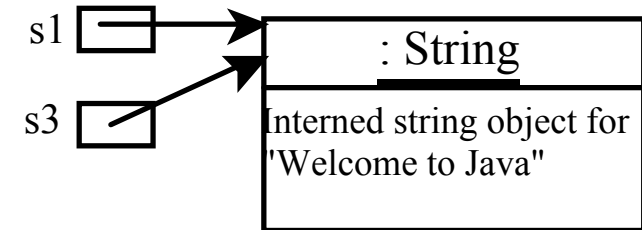


Trace Code

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

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

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

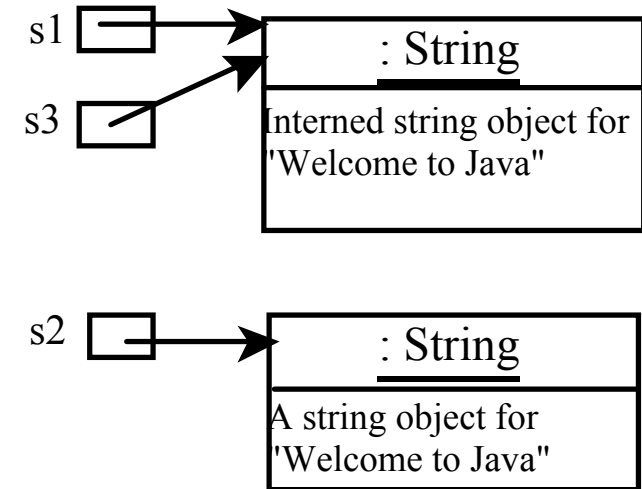


Trace Code

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

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

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

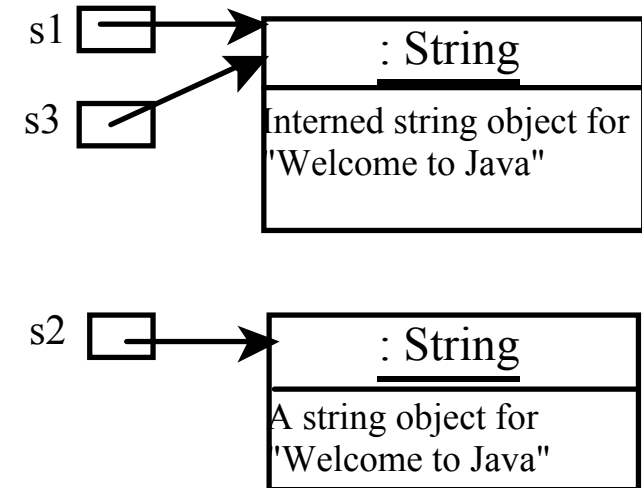


Interned string – a *unique instance* is used for *string literals* with the *same character sequence*:

- A new object is not created if the interned string has already been created
- A new object is created if the `new` operator is used.

Trace Code

```
String s1 = "Welcome to Java";  
String s2 = new String("Welcome to Java");  
String s3 = "Welcome to Java";
```



Interned string – a *unique instance* is used for *string literals* with the *same character sequence*:

- A new object is not created if the interned string has already been created
- A new object is created if the `new` operator is used.

In the above example:

- `s1 == s3` is true
- `s1 == s2` and `s2 == s3` are both false
- `s1.equals(s2)`, `s1.equals(s3)` and `s2.equals(s3)` are all true

StringBuilder Class

The `StringBuilder` class is an alternative to the `String` class.

`StringBuilder` is more flexible than `String`, and allows you to add, insert, or append new contents, whereas the value of a `String` object is fixed once the string is created.

In general, a `StringBuilder` can be used wherever a string is used:

- Note: If a string does not require any changes, use `String` rather than `StringBuilder` (to take advantage of interned strings).

StringBuilder Class

The `StringBuilder` class provides several overloaded methods to **append** `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long`, and `String` into a string builder.

For example, the following code appends strings and characters into a `StringBuilder` object `sb` to form a new string, "Welcome to Java".

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

StringBuilder Class

The `StringBuilder` class also contains overloaded methods to `insert` `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long`, and `String` into a string builder.

Suppose `sb` contains `"Welcome to Java"`. Consider the following code:

```
sb.insert(11, "HTML and ");
```

This code inserts `"HTML and "` at position 11 in `sb` (just before the `'J'`), now `sb` is set to `"Welcome to HTML and Java"`.

StringBuilder Class

You can also delete characters from a string in the builder using the `delete` methods, reverse the string using the `reverse` method, replace characters using the `replace` method, or set a new character in a string using the `setCharAt` method.

For example, suppose `sb` contains "Welcome to Java" before each of the following methods is applied:

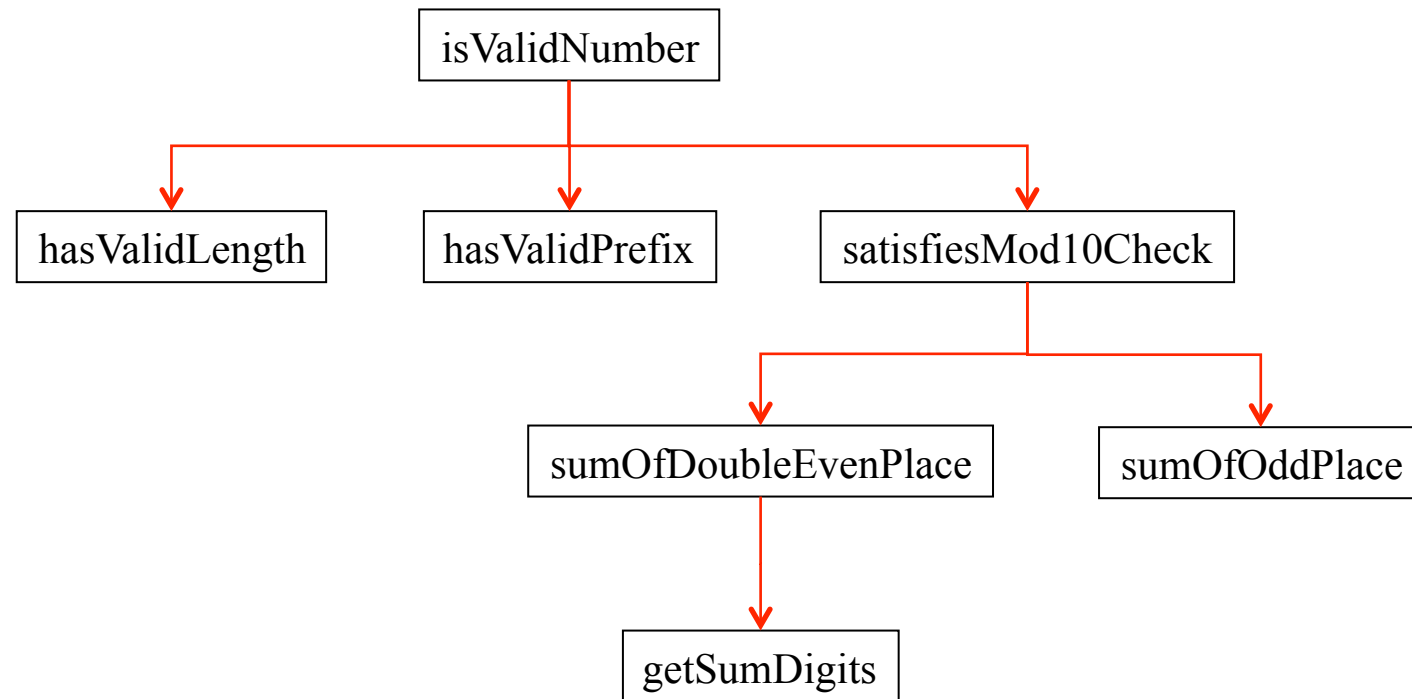
- `sb.deleteCharAt(8)` changes the builder to "Welcome o Java".
- `sb.reverse()` changes the builder to "avaJ ot emocleW".
- `sb.replace(11, 15, "HTML")` changes the builder to "Welcome to HTML".
- `sb.setCharAt(0, 'w')` sets the builder to "welcome to Java".

See the Java API for more details.

Procedural Programming Paradigm

From Chapter 6 – checking if a credit card number is valid...

Using stepwise refinement, a number of methods are defined. Methods invoke other methods, dependencies exist between methods (by design):



Procedural Programming Paradigm

Procedural programming is based upon the concept of procedure calls.

A procedure simply contains a series of computational steps to be carried out.

Any given procedure might be called at any point during a program's execution, including by other procedures (or by itself).

In small programs, this approach works fine.

But as specifications become more complicated and the size of programs grows, small changes to one part of the program can greatly affect (many) other parts.

Procedural vs. Object-Orientated Programming Paradigms

The traditional procedural programming paradigm is action-driven, and data are separated from actions:

- Suppose you wish to associate the balance, interest rate, repayment dates with a loan – there is no good way to tie these properties to a loan without using objects.

The object-oriented paradigm integrates data and methods together into objects:

- To tie a balance to a loan, you can define a loan class with a balance along with the loan's other properties as data fields (interest rate, repayment dates etc.).
- Actions are defined along with the data in objects – actions that are particular to the object's data (e.g. deposit, compute interest due, etc.).

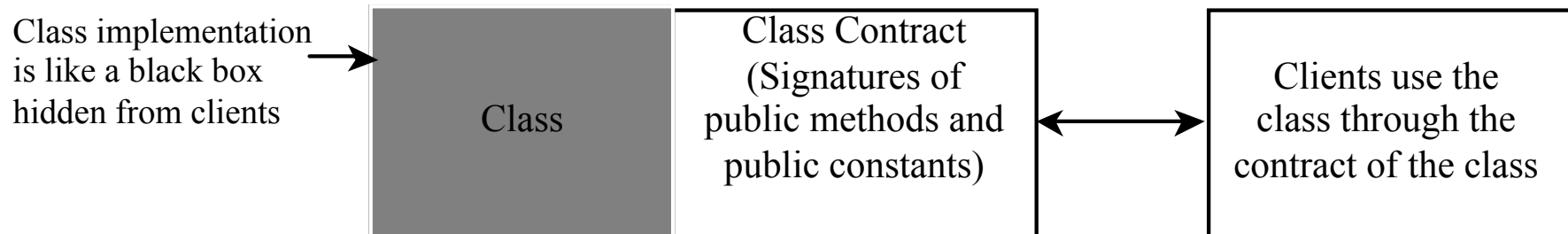
Using objects improves software reusability and makes programs easier to develop, easier to maintain, and reduces bugs in the code.

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 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 Relationships

To design classes, you need to explore the *relationships* among classes.

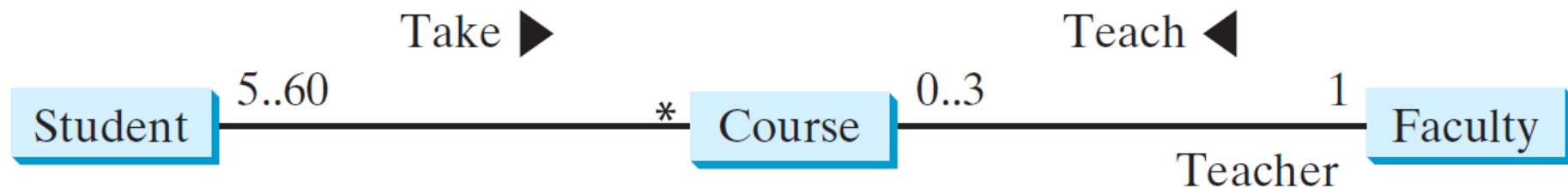
The common relationships among classes are *association*, *aggregation*, *composition*, and *inheritance*.

Here we explore association, aggregation, and composition. The inheritance relationship will be introduced in the next chapter.

Association

Association is a general binary relationship that describes an activity between two classes.

For example, consider a course, the students taking the course, and the faculty teaching the course:



- This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from five to sixty students, and a course is taught by only one faculty member.
- A student taking a course is an association between the `Student` class and the `Course` class, and a faculty member teaching a course is an association between the `Faculty` class and the `Course` class.

Association

Associations are implemented by using data fields and methods.

```
public class Student {  
    private Course[]  
        courseList;  
  
    public void addCourse(  
        Course s) { ... }  
}
```

```
public class Course {  
    private Student[]  
        classList;  
    private Faculty faculty;  
  
    public void addStudent(  
        Student s) { ... }  
  
    public void setFaculty(  
        Faculty faculty) { ... }  
}
```

```
public class Faculty {  
    private Course[]  
        courseList;  
  
    public void addCourse(  
        Course c) { ... }  
}
```

- Relation “a student takes a course” is implemented using the `addCourse` method in the `Student` class and the `addStudent` method in the `Course` class.
- The relation “a faculty teaches a course” is implemented using the `addCourse` method in the `Faculty` class and the `setFaculty` method in the `Course` class.
- Note: There are many possible ways to implement relationships. For example, the student and faculty information in the `Course` class can be omitted, since they are already in the `Student` and `Faculty` class.

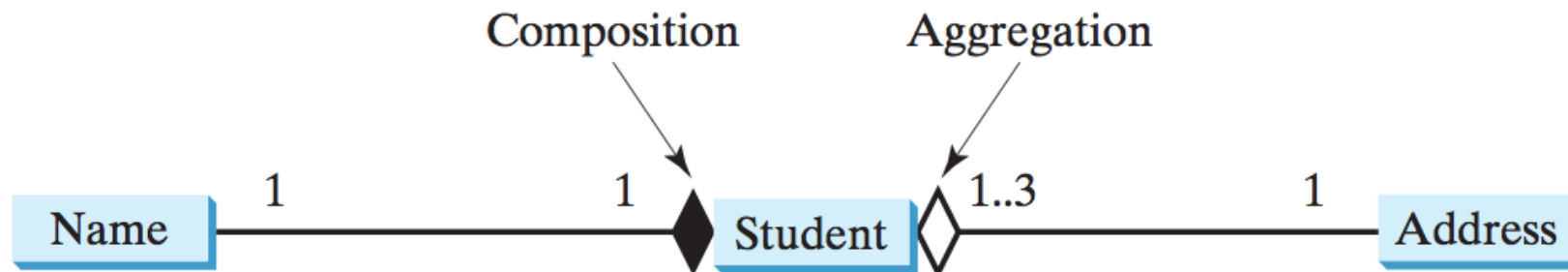
Aggregation and Composition

Aggregation is a special form of association. It represents an *ownership* relationship between two objects and models *has-a* relationships.

- The owner object is called an *aggregating object* and its class an *aggregating class*.
- The subject object is called an *aggregated object* and its class an *aggregated class*.

An object can be owned by several other aggregating objects. If an object is exclusively owned by an aggregating object, the relationship is referred to as a *composition*. For example:

- “a student *has an* address” is an *aggregation* relationship between a student and her address, since an address can be shared by several students.
- “a student *has a* name” is a *composition* relationship between a student and her name, since each student has one name.



Aggregation and Composition

An aggregation/composition relationship is 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

For example:

- The relations “a student has a name” and “a student has an address” are implemented in the data fields `name` and `address` in the `Student` class.

Aggregation and composition relationships are represented using classes in the same way.

Next Lectures...

This and previous lectures:

- Chapter 9 – introducing classes and objects
- Chapter 10 – strings, object-oriented vs. procedural programming paradigms, class relationships

Next lectures:

- Chapter 11 – inheritance and polymorphism