COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

OBJECT-ORIENTED PROGRAMMING

COMP2311

Instructor :Murad Njoum
Office : Masri322

Chapter 10 Thinking in Objects and Strings -Revision

Constructing Strings

String newString = new String(stringLiteral);

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

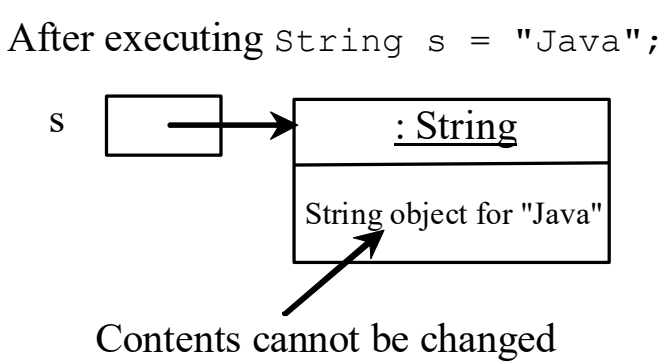
A String object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

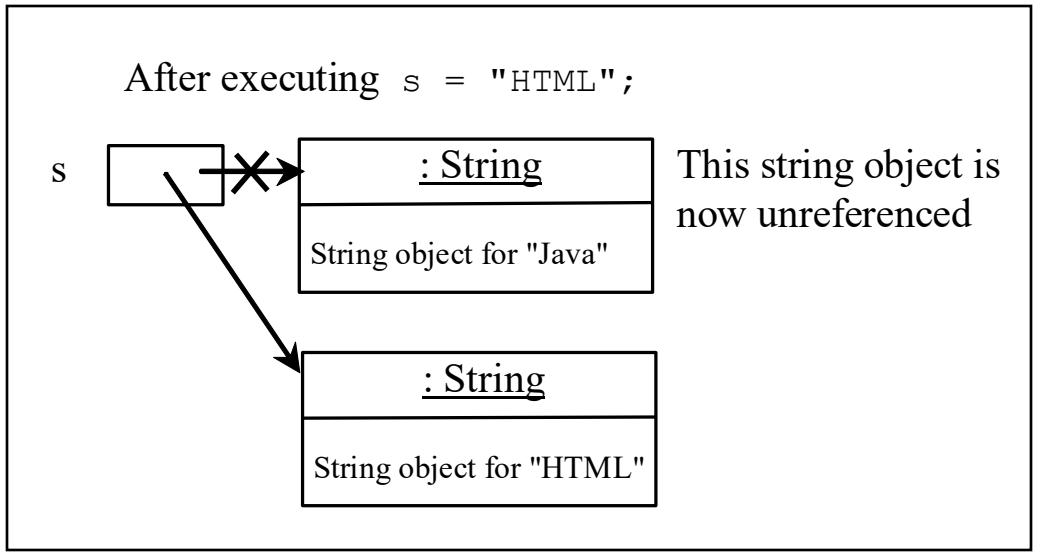
```
String s = "Java";
s = "HTML";
```

Trace Code

`String s = "Java";`

`s = "HTML";`



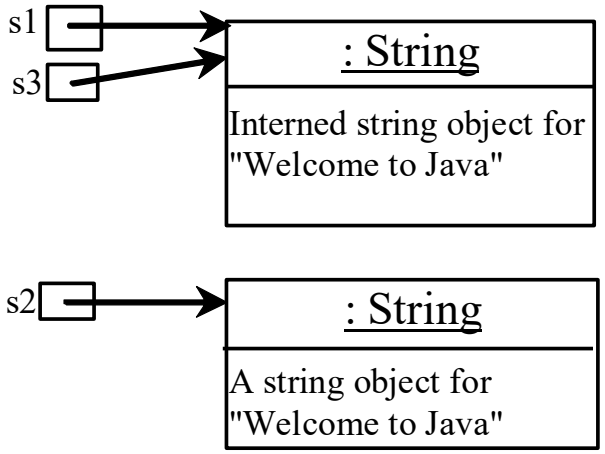


Interned Strings

Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a unique instance for string literals with the same character sequence. Such an instance is called interned. For example, the following statements:

Examples

```
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));
```



display

s1 == s2 is false

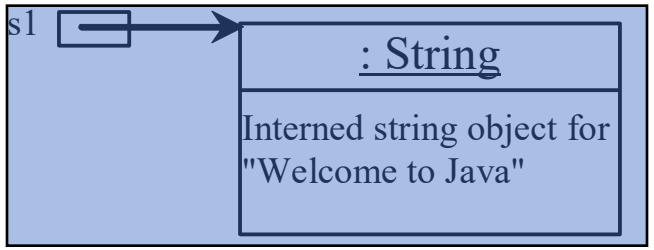
s1 == s3 is true

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.

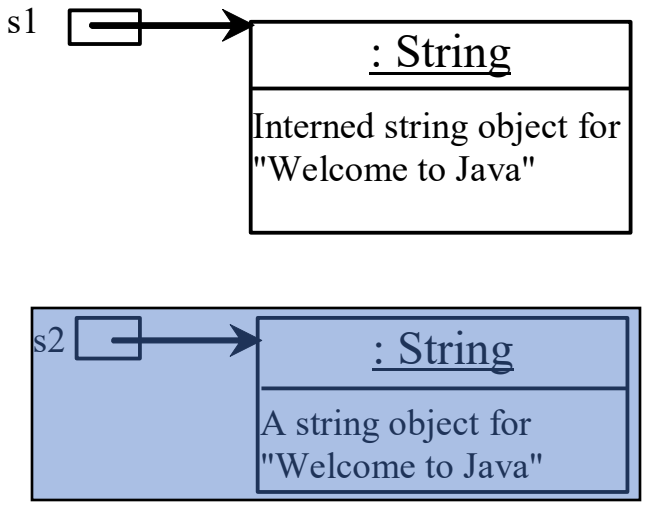
Trace Code

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

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

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



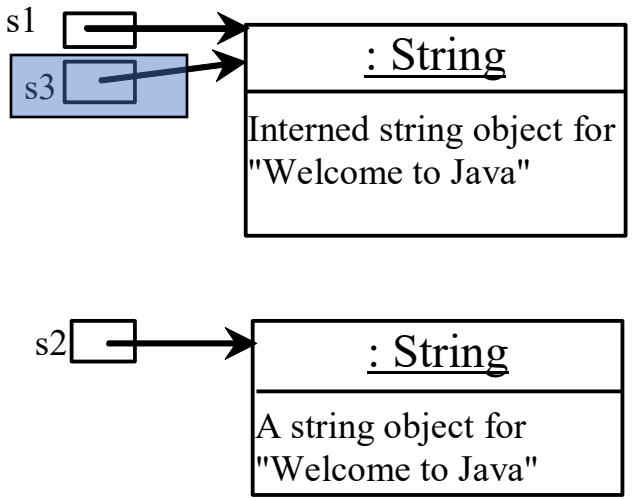
Trace Code

```
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";
```



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.

Examples

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

"Welcomel".replaceAll("el", "AB") returns a new string, WABcomAB.

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

```
Java HTML   Perl hello hi
```

Matching, Replacing and Splitting by Patterns

The following statement splits the string into an array of strings delimited by some punctuation marks.

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);
```

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'.

String.valueOf(tokens[0]).

`StringBuilder` and `StringBuffer`

→The `StringBuilder/StringBuffer` class is an **alternative** to the `String` class.

→In general, a 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

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

Examples

```
StringBuilder stringBuilder = new StringBuilder("Welcome Java");
```

Welcome JavaJava
Welcome JavHTML and aJava
Welcome Java
Welcome ava
ava emocleW
ava emocleWHTML

stringBuilder.append("Java");
stringBuilder.insert(11, "HTML and ");
stringBuilder.delete(8, 21) changes the builder to Welcome Java.
stringBuilder.deleteCharAt(8)
stringBuilder.reverse() changes the builder to avaJ ot emocleW.
stringBuilder.replace(11, 15, "HTML")
changes the builder to Welcome to HTML.
stringBuilder.setCharAt(0, 'w') sets the builder to welcome to Java.

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.

Regular Expressions

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. Regular expression is a powerful tool for string manipulations. You can use regular expressions for matching, replacing, and splitting strings.

Matching Strings

```
"Java".matches("Java");
"Java".equals("Java");
```

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

Regular Expression Syntax

"Java".matches("J..a");

"Java".matches("J(av|ba)a");

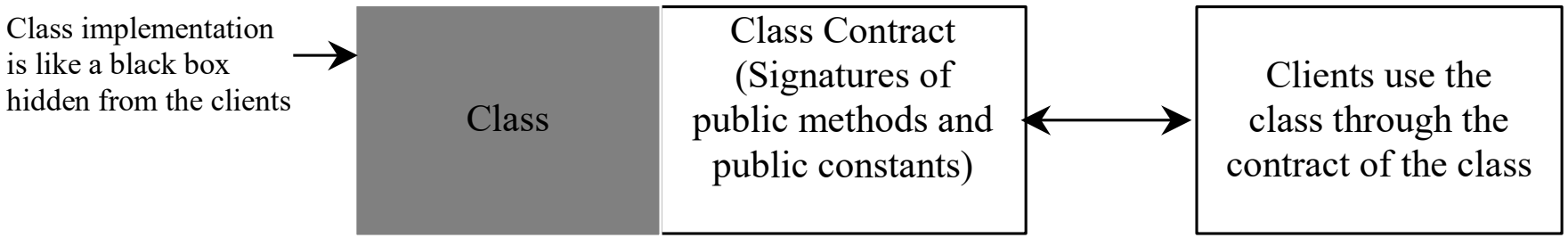
Regular Expression	*Matches*	*Example*
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab|cd)	ab or cd	ten matches t(en|im)
[abc]	a, b, or c	Java matches Ja[uvwx]a
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]
\d	a digit, same as [0-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java1 matches "[\\w]ava[\\w]"
\W	a non-word character	$Java matches "[\\W][\\w]ava"
\s	a whitespace character	"Java 2" matches "Java\\s2"
\S	a non-whitespace char	Java matches "[\\S]ava"
*p**	zero or more occurrences of pattern *p*	aaaabb matches "a*bb" ababab matches "(ab)*"
p+	one or more occurrences of pattern *p*	a matches "a+b*" able matches "(ab)+.*"
p?	zero or one occurrence of pattern *p*	Java matches "J?Java" Java matches "J?ava"
p{n}	exactly n occurrences of pattern *p*	Java matches "Ja{1}.*" Java does not match ".{2}"
p{n,}	at least n occurrences of pattern *p*	aaaa matches "a{1,}" a does not match "a{2,}"
p{n,m}	between n and m occurrences (inclusive)	aaaa matches "a{1,9}" abb does not match "a{2,9}bb"

Thinking in Objects

You see the advantages of object-oriented programming from the preceding chapter. This chapter will demonstrate how to solve problems using the object-oriented paradigm.

Class Abstraction and Encapsulation

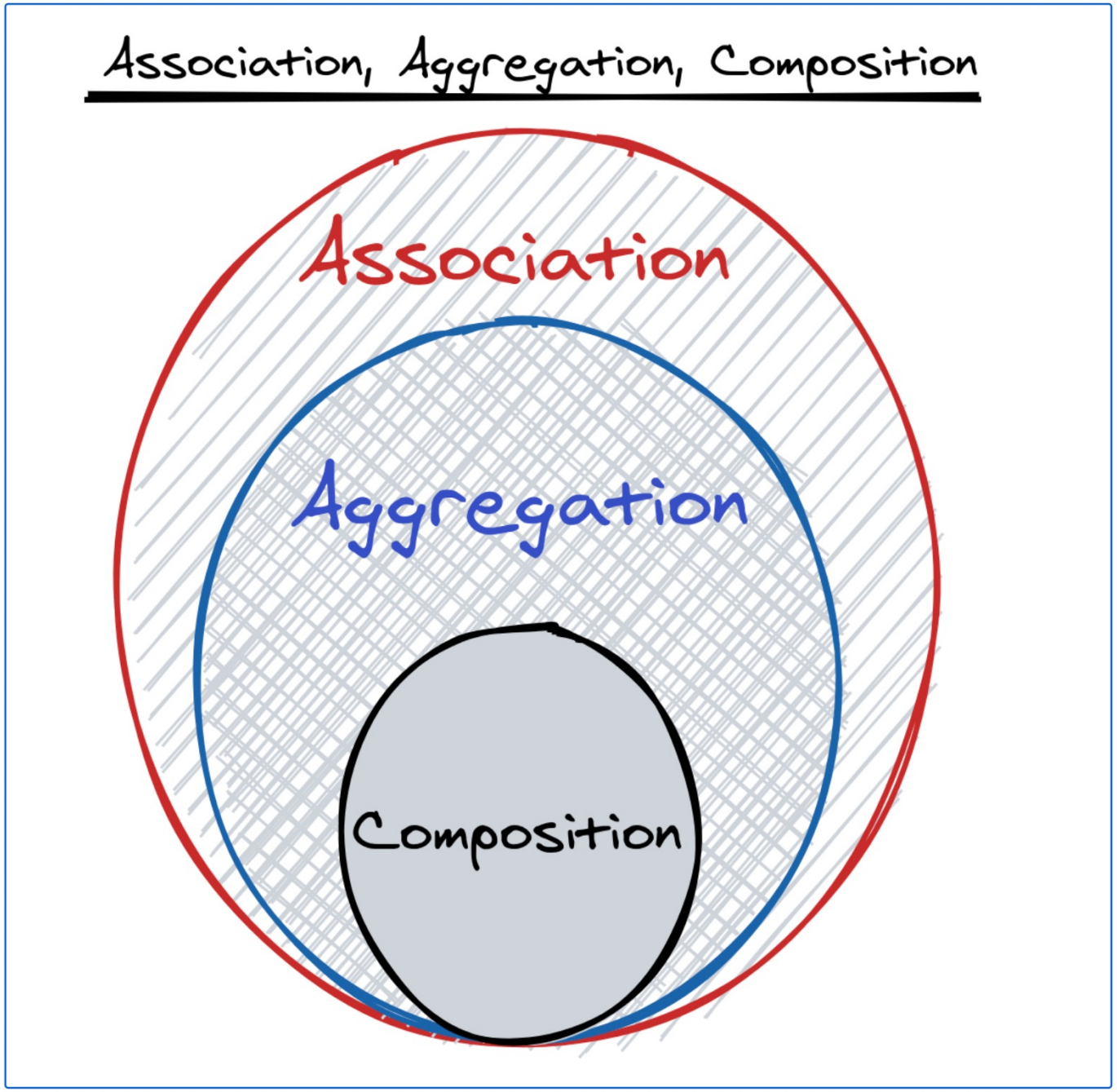
Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is **encapsulated** and hidden from the user.



Object-Oriented Thinking

Chapters 1-8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. The studies of these techniques lay a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 **using the object-oriented approach**. From the improvements, you will gain the insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.

Relations :

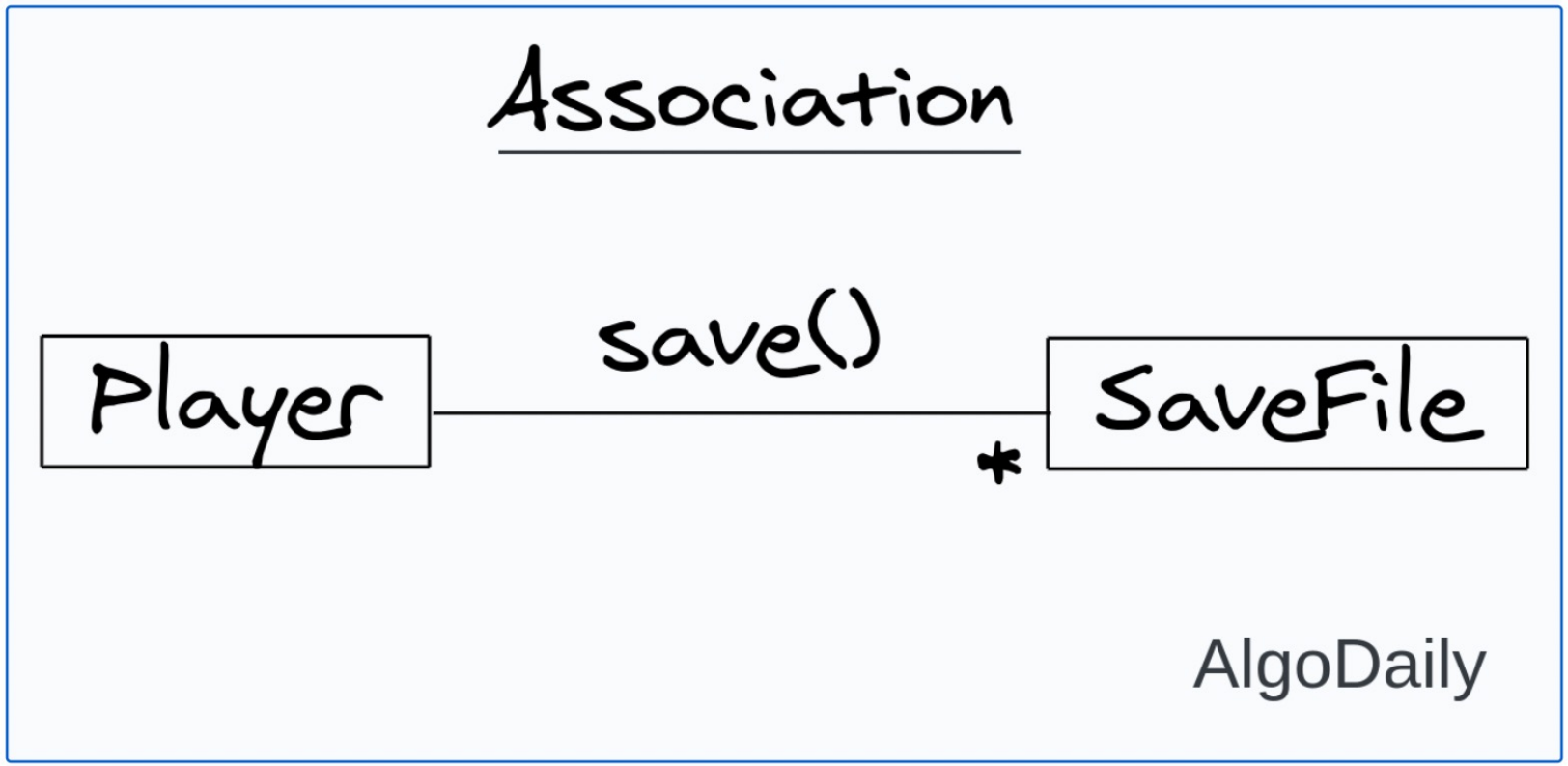

Association, Aggregation, Composition
Association
Aggregation
Composition

Association

Whenever two classes are connected to each other, an association relationship link can be used.

(No ownership , no lifetime dependency)

You can use a simple name for the relationship close to the line. For example, in a game, a player will have a lot of save files. If we consider Player and SaveFile as classes, then we can create an association link for them like below:



Aggregation

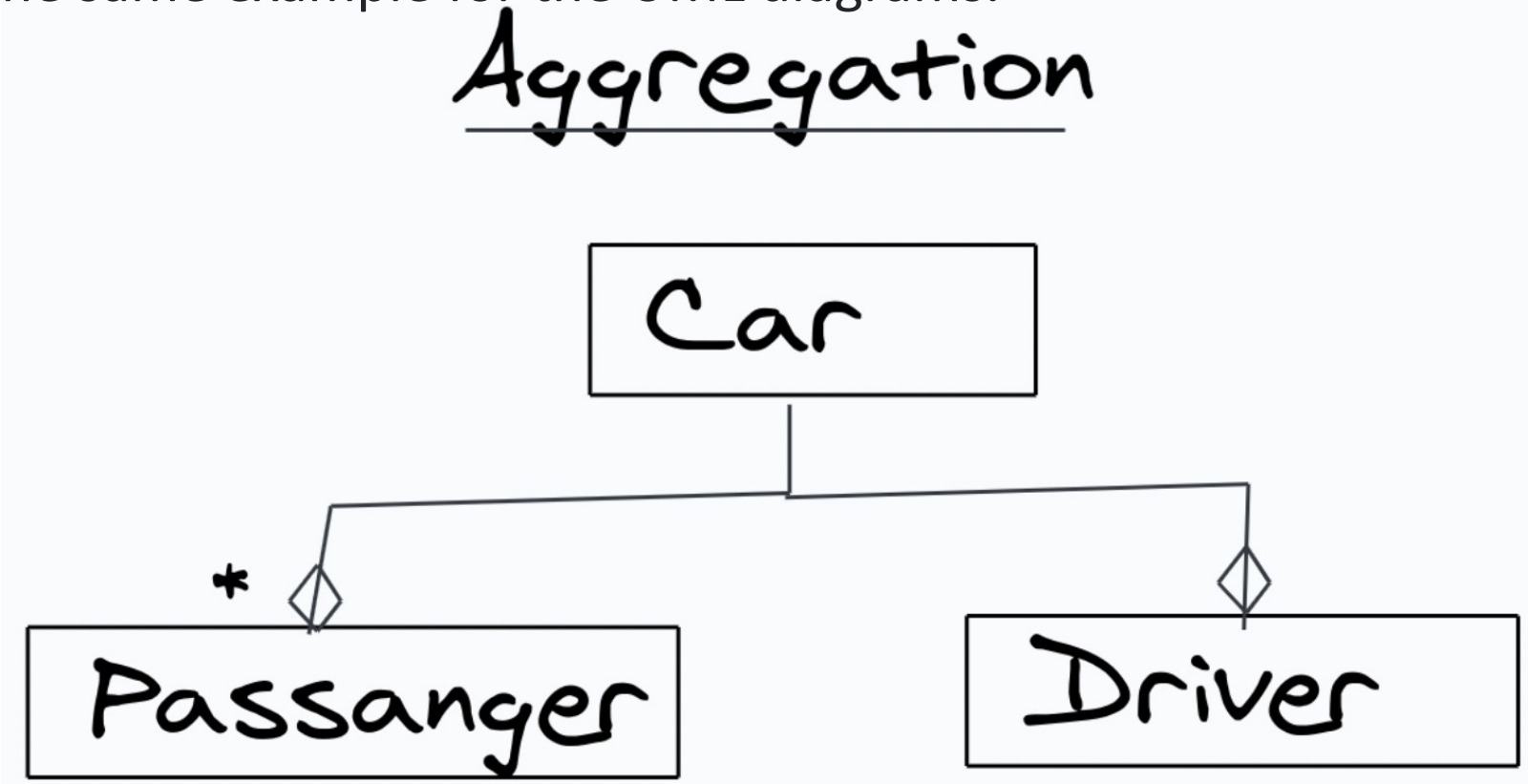
With aggregation, an object will always be referenced by other objects.

(One owner instance but no lifetime dependency)

This can be shown by an open diamond (a diamond without any fill color) in the UML.

Previously we described aggregation and composition using the Vehicle class example.

We will use the same example for the UML diagrams.

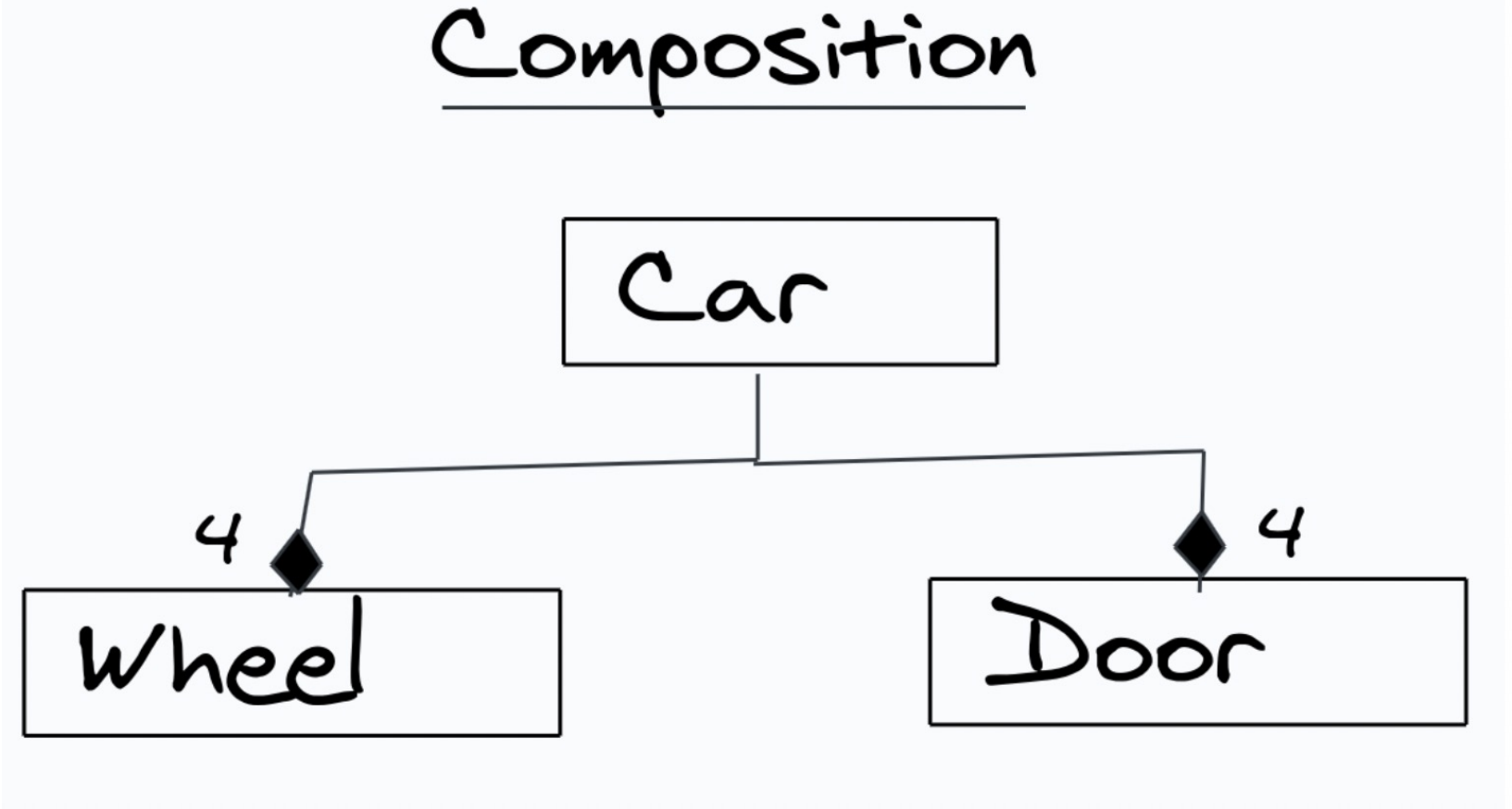


Composition

If an object only contains one other object such that their lives are bound together,

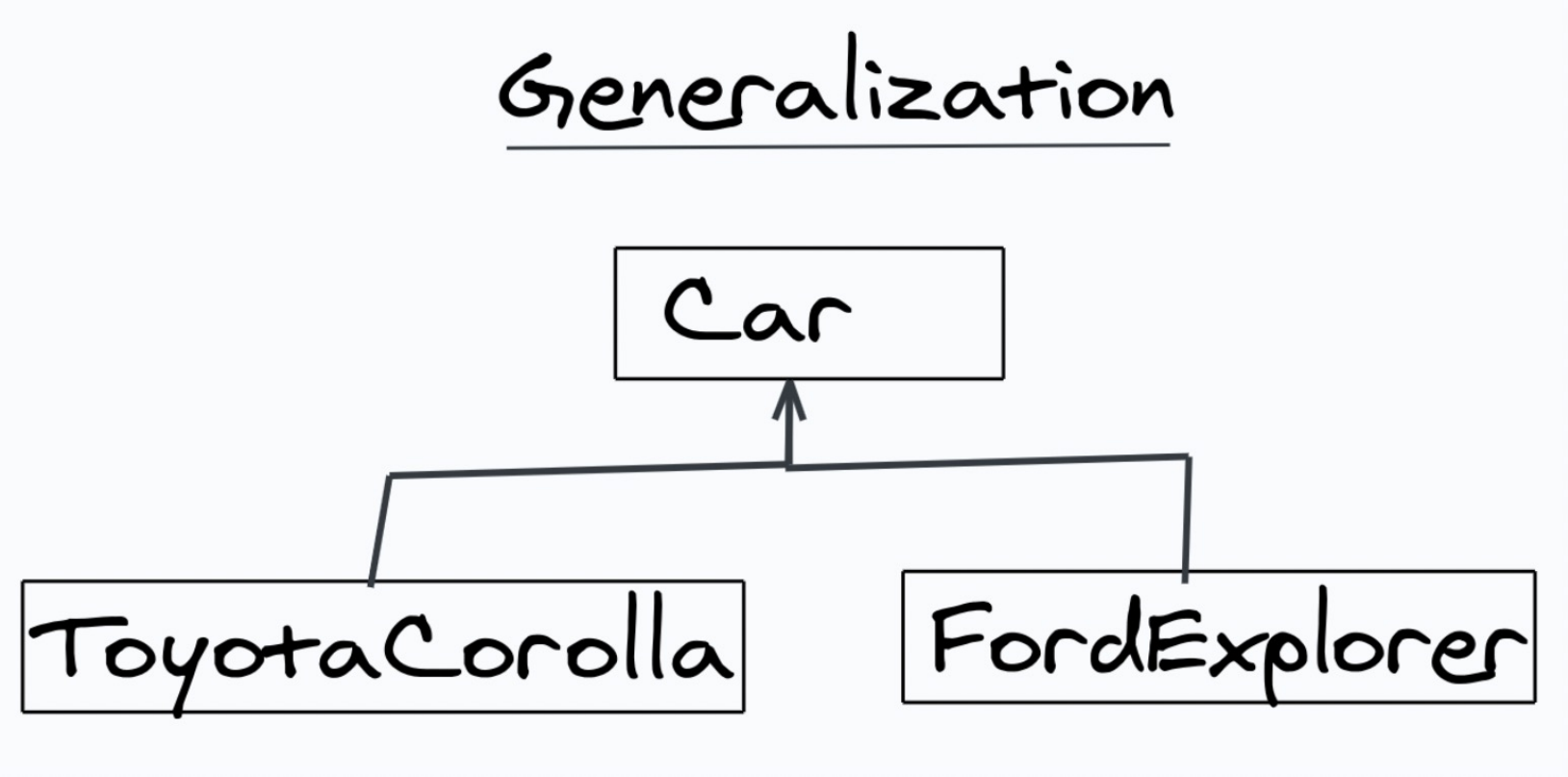
(One owner instance and lifetime child instance dependent on lifetime of owner instance)

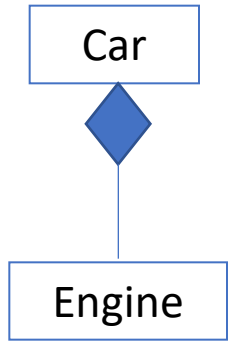
then we can show this relationship with the composition arrow in UML.



Generalization

Generalization is a synonym of inheritance in the world of OOP. When a **class is inherited from another class**, then we can show this inheritance relationship with a simple arrow from the child class to the parent class.





Part of

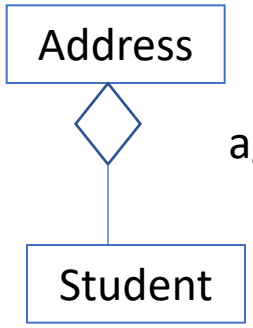
composition

```
class Car {
private final Engine engine;
 Car(){
    engine=new Engine();
    }//final initialized once
}

class Engine {
private String type;
}
```

…

```
Car car=new Car();
```

…



Has-a

aggregation

```
class Student {
private Address address;
 Student(Address addr){
    address=addr;
    }
}
  class Address {
  String city;
  String state;
   Address(String city, String state){
      this.city=city; this.state=state;
      }
  }
```

…

```
Student student=new Student();
```

…

- **Create instance:**

(engine automatically created once), student has passed parameters from other methods

- **Delete instance:** delete car instance ,automatically engine instance deleted and can't passed to other car instance, but inf class student deleted then address can be passed to other students

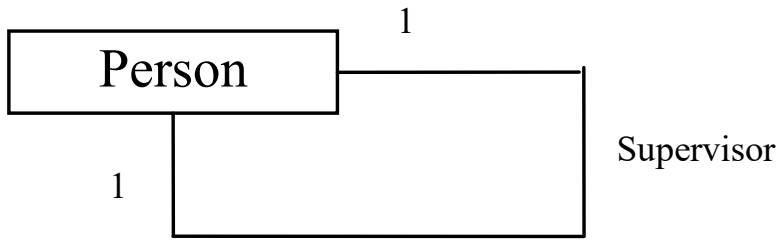
Overloading Constructors

- If you create a class from which you instantiate objects, Java automatically provides a constructor
- But, if you create your own constructor, the automatically created constructor no longer exists
- As with other methods, you can overload constructors
  - Overloading constructors provides a way to create objects with or without initial arguments, as needed

Aggregation Between Same Class

Aggregation may exist between objects of the same class.

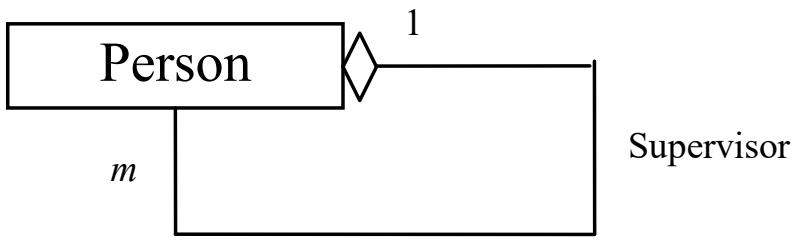
For example, a person may have a supervisor.




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

Aggregation Between Same Class

What happens if a person has several supervisors?




```
public class Person {
  ...
  private Person[] supervisors;
}
```

Wrapper Classes

- Boolean
- Character
- Short
- Byte
- Integer
- Long
- Float
- Double

NOTE:

(1) The wrapper classes **do not have no-arg constructors**.

(2) The instances of all wrapper classes are **immutable,** i.e., their internal values cannot be changed once the objects are created.

The `Integer` and `Double` 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

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

The `Integer` Class and the `Double` Class

- Constructors
- Class Constants `MAX_VALUE`, `MIN_VALUE`
- Conversion Methods

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**. 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.
- The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45),

and the maximum double floating-point number (1.79769313486231570e+308d).

Conversion Methods

Each numeric wrapper class implements the abstract methods doubleValue, floatValue, intValue, longValue, and shortValue, which are defined in the Number class. These methods “convert” objects **into primitive type** values.

The Static valueOf Methods

The numeric wrapper classes have a useful class method, **valueOf(String s).** This method creates a new object initialized to the value represented by the specified string. For example:

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

The Methods for Parsing Strings into Numbers

You have used the **parseInt** method in the **Integer class** to parse a **numeric string** into an int value

and the parseDouble method in the Double class to parse a numeric string into a double value.

Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 **allows primitive type and wrapper** classes to be converted automatically.
For example, the following statement in (a) can be simplified as in (b):

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

(a)

Equivalent

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

New JDK 1.5 boxing

(b)

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

Unboxing

Vis versa is also true.

BigInteger and BigDecimal

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.

BigInteger and BigDecimal

```
BigInteger a = new BigInteger("92233720368547758O7");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 92233720368547758O7 * 2
System.out.println(c);
```

LargeFactorial Run

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

```
package test;

import java.util.Scanner;
import java.math.*;

public class LargeFactorial {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int n = input.nextInt();
    System.out.println(n + "! is \n" + factorial(n));
 input.close();
}

  public static BigInteger factorial(long n) {
    BigInteger result = BigInteger.ONE; // Assign 1 to
result
    for (int i = 1; i <= n; i++) // Multiply each i
      result = result.multiply(BigInteger.valueOf(i));

    return result;
  }
}
```