AP Computer Science B

Java Object-Oriented Programming [Ver. 3.0]

Unit 4: Object-Oriented Programming

CHAPTER 11: OBJECT-ORIENTED THINKING

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

- •Object-Thinking: Design of a Class
- •How to Design a Good Class
- •Completeness and Design Conventions
- •Class Loading Time and Data Memory Model
- •has_A Relationship
- •Is_A Relathionship
- •BigInteger/BigDecimal/String/StringBuilder/StringBuffer Classes

Object-Thinking: Design of a Class

LECTURE 1

Object-Oriented Thinking

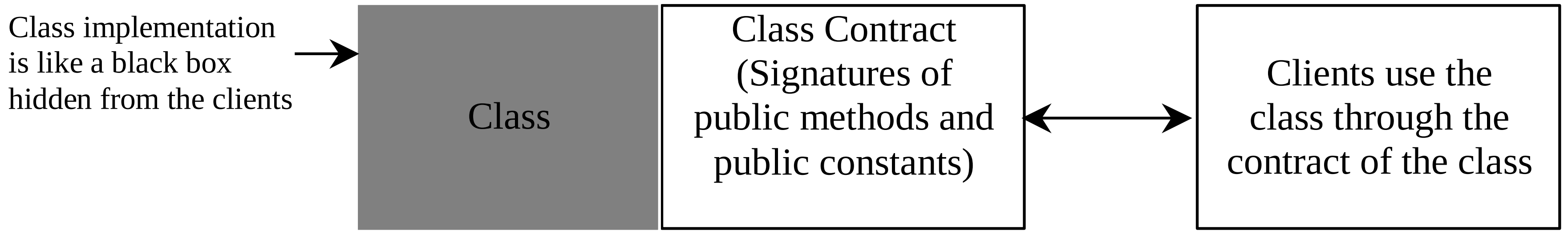
- Part 1-Chapters 1-9 introduced fundamental programming techniques for problem solving using loops, methods, and arrays.
- The studies of these techniques lay a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software.

Object-Oriented Thinking

- This chapter reviews chapter 10 and improves the solution for a problem introduced in Part-1 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.

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.



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 a Class: Coherence

- (**Coherence**) A class should describe a **single entity**, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.

e.g. Student, Subject, ScoreSheet, Card, Deck, and Hand

Designing a Class, cont.

- (**Separating responsibilities**) A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
- The classes **String**, **StringBuilder**, and **StringBuffer** all deal with strings, for example, but have different responsibilities. The String class deals with immutable strings, the StringBuilder class is for creating mutable strings, and the StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.

Designing a Class, cont.

- Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes **no restrictions** on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.

Designing a Class, cont.

- Provide a public no-arg constructor and override the **equals** method and the **toString** method defined in the Object class whenever possible.
- Overriding standard methods inherited from Object class.

Designing a Class, cont.

- Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods.
- Always place the data declaration before the constructor, and place constructors before methods.
- Always provide a constructor and initialize variables to avoid programming errors.

How to Design a Good Class

LECTURE 2

Guidelines for Class Design

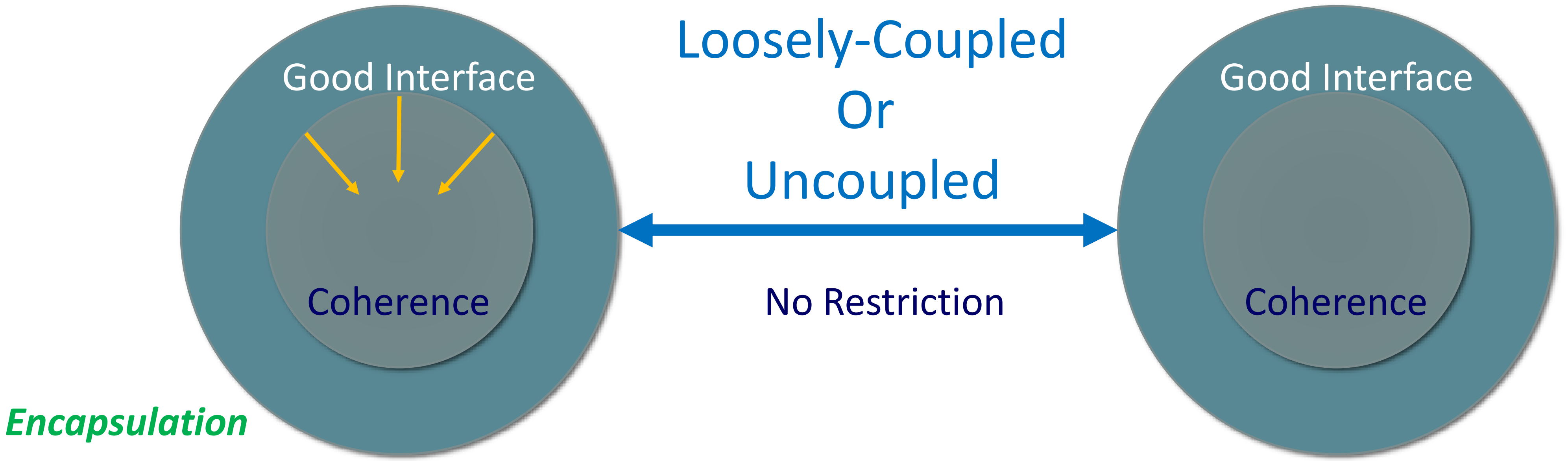
- Good design of individual classes is crucial to good overall system design. A well-designed class is **more re-usable** in different contexts, and **more modifiable** for future versions of software.
- Here, we'll look at some general class design guidelines, as well as some tips for specific languages, like **C++** or **Java**.

(Chapter 9’s class design guidelines on technical issue, this chapter is on design styles)

General goals for building a good class

- Having a **good usable interface** (**Information Hiding**)
- **Implementation objectives**, like efficient algorithms and convenient/simple coding
- Separation of implementation from interface!! (**Encapsulation**)
- Improves **modifiability** and **maintainability** for future versions (**re-usability**)
- **Decreases *coupling* between classes**, i.e. the amount of dependency between classes (will changing one class require changes to another?) (**no restrictions**)
- Can re-work a class inside, without changing its interface, for outside interactions
- Consider how much the automobile has advanced technologically since its invention. Yet the basic interface remains the same -- steering wheel, gas pedal, brake pedal, etc.

Good Class Design



One Class One Purpose

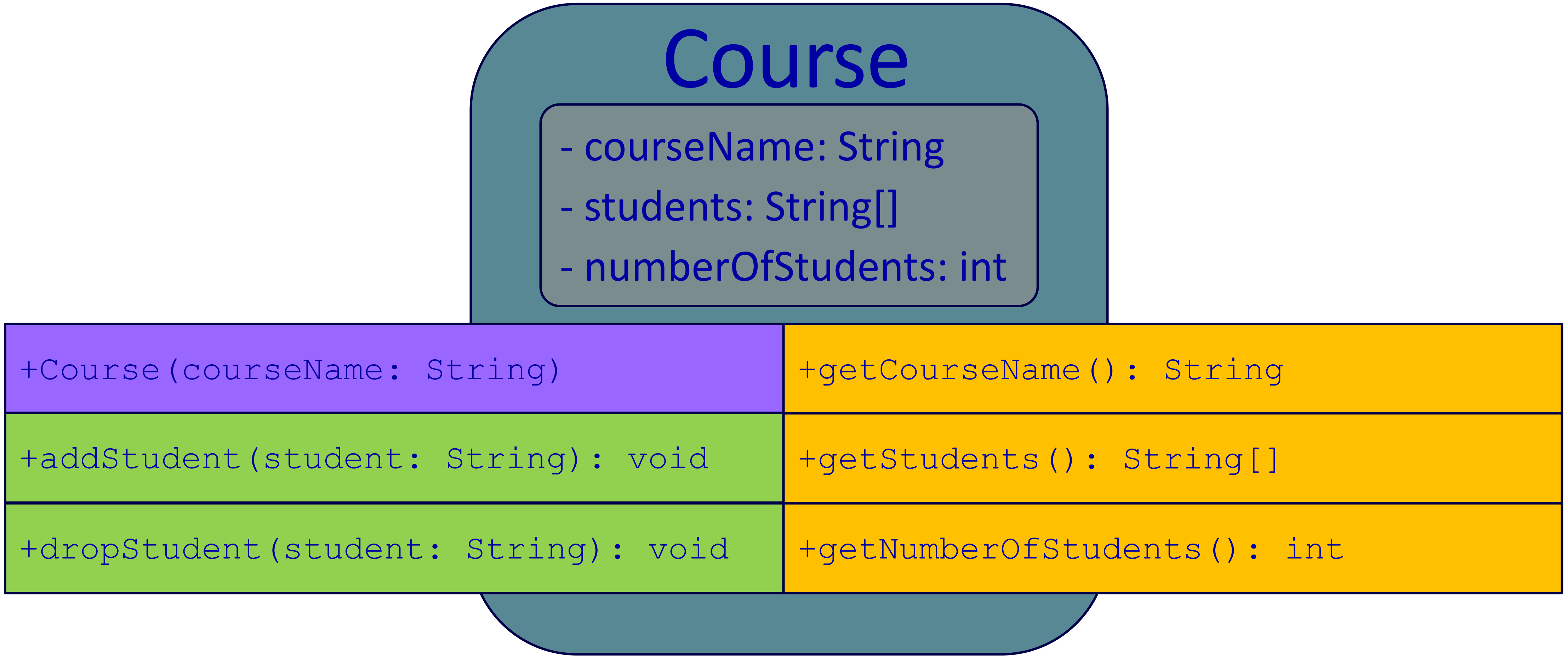
Designing a good class interface

By interface, we are talking about what the class user sees. This is the public section of the class.

- **Cohesion** (or *coherence*): one class single abstraction
- **Completeness**: A class should support all important operations
- **Convenience**:
  1. User-friendly
  2. API-Oriented (written like API)
  3. Systematic
- **Clarity**: No confusion or ambiguous interpretations
- **Consistency**
  1. Operations in a class will be most clear if they are consistent with each other.
  2. Naming conventions (toString, compareTo, equals, isLetter, and etc.)

Course Class

One Abstraction: Course Information



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

The Course Class

Course	
-courseName: String	The name of the course.
-students: String[]	An array to store the students for the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course.
+dropStudent(student: String): void	Drops a student from the course.
+getStudents(): String[]	Returns the students in the course.
+getNumberOfStudents(): int	Returns the number of students in the course.

Demonstration Program

COURSE.JAVA/TESTCOURSE.JAVA

Completeness and Design Conventions

LECTURE 3

Class Design

Class Data Fields:
- Constants
- Class Variables

Member Data Fields:
- Instance Variables

Constructors:
- no-arg
- Long-form

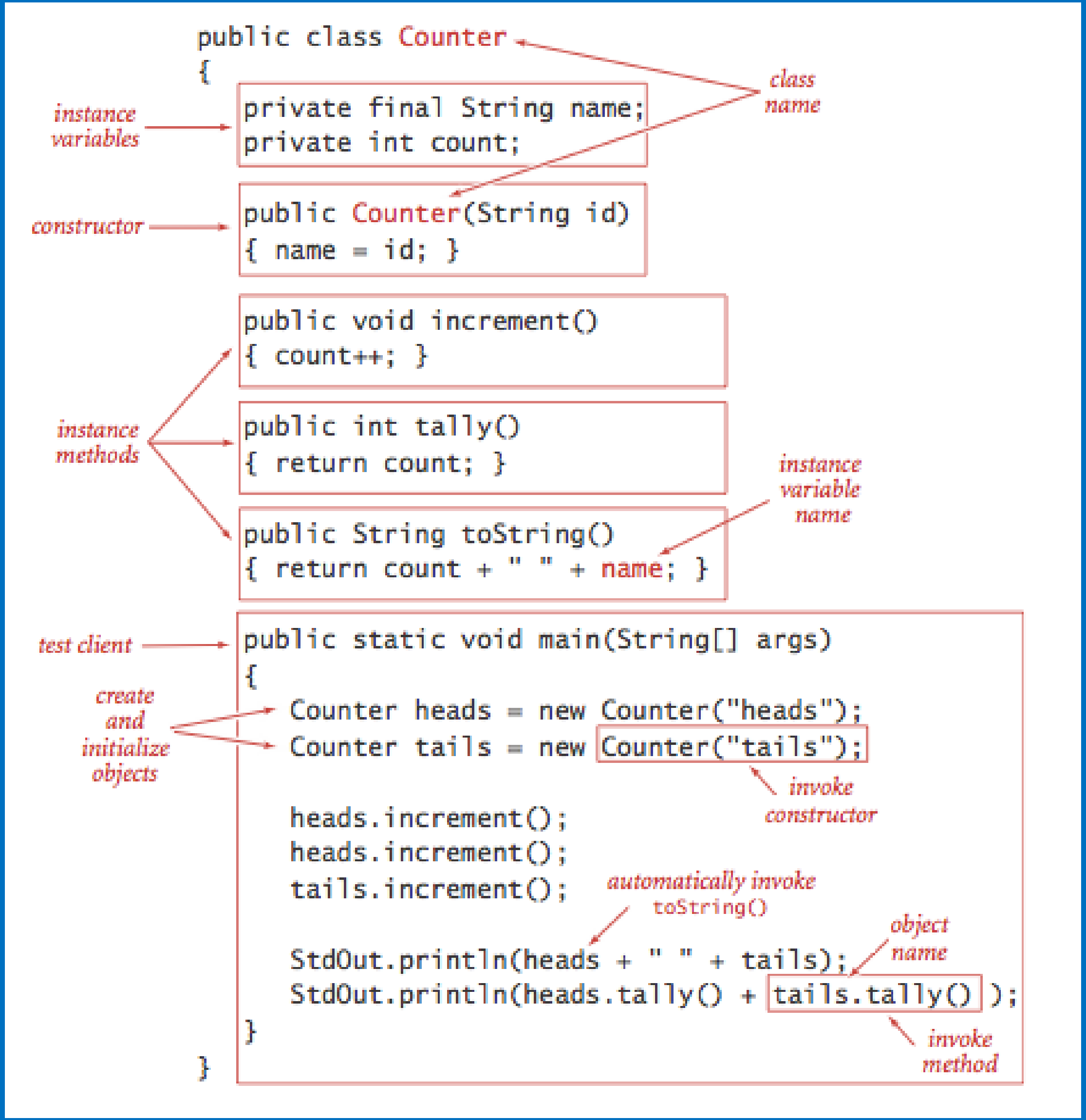
Getters/Setters Method:
- no-arg
- Long-form

Inherited Methods:
- toString()
- Equals()
- compareTo()
- next(), hasNext(), remove()

Derived Data Fields:
- getDerivedData()

Utility Methods:
- Static Methods

Class Writing Styles
(Not Syntax, Design Conventions)



```java
public class Counter
{
   private final String name;
   private int count;

   public Counter(String id)
   { name = id; }

   public void increment()
   { count++; }

   public int tally()
   { return count; }

   public String toString()
   { return count + " " + name; }

   public static void main(String[] args)
   {
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");

      heads.increment();
      heads.increment();
      tails.increment();

      StdOut.println(heads + " " + tails);
      StdOut.println(heads.tally() + tails.tally() );
   }
}
```

Completeness

How to create it? How to get it? How to change it? And, how to show it?

Constructor Parameter Type Manipulation (Overloading of Constructor Method):

// Make user easier to use.

Complex(), Complex(double r), Complex(double r, double i)

Accessor/Mutator:

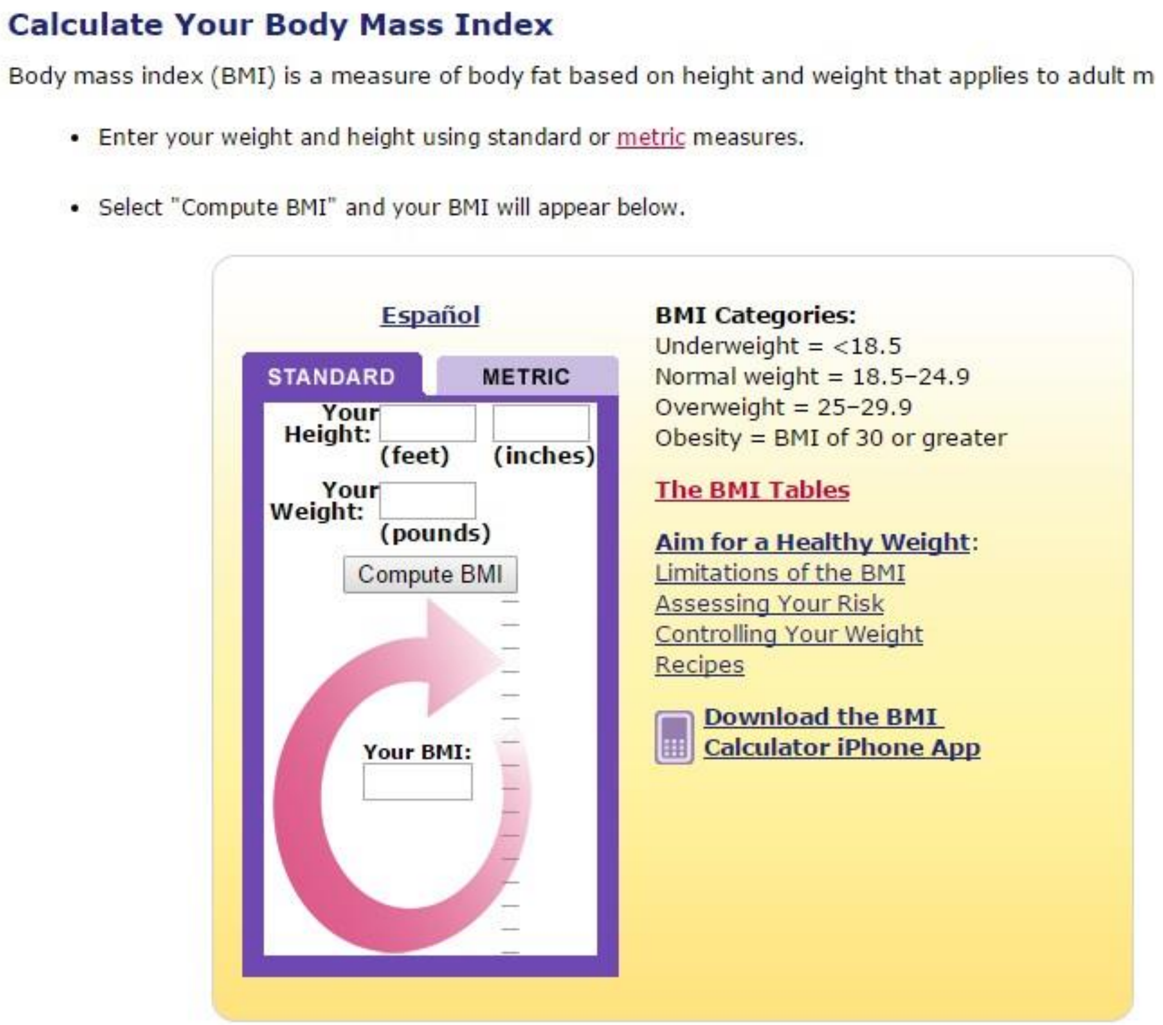
getR(), getI(), setR(), setI(), set(double r, double i);

Comparators:

.equals(), Complex.equals(Complex c1, Complex c2)

To String Method:

.toString(), Complex.toString(Complex c)



Calculate Your Body Mass Index

Body mass index (BMI) is a measure of body fat based on height and weight that applies to adult m

- Enter your weight and height using standard or metric measures.
- Select "Compute BMI" and your BMI will appear below.

BMI Categories:
Underweight = <18.5
Normal weight = 18.5–24.9
Overweight = 25–29.9
Obesity = BMI of 30 or greater

The BMI Tables

Aim for a Healthy Weight:
Limitations of the BMI
Assessing Your Risk
Controlling Your Weight
Recipes

Download the BMI Calculator iPhone App

BMI calculation

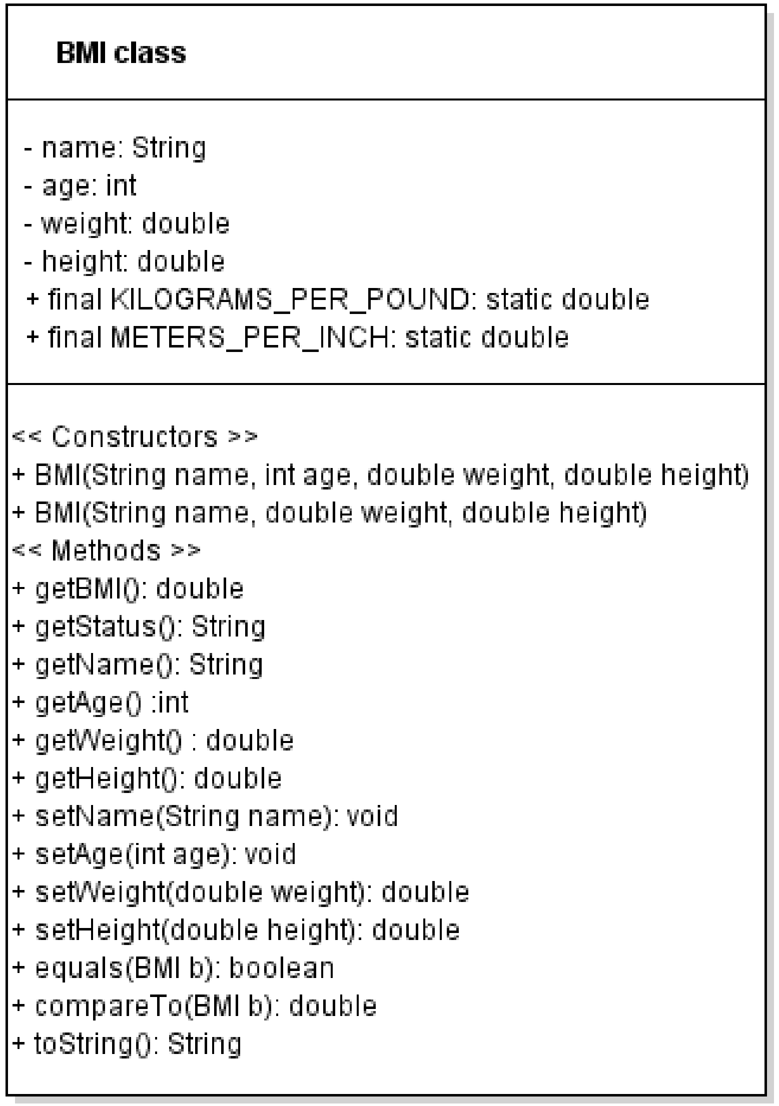
http://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmicalc.htm

The BMI Class

Incomplete Version

The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

BMI	
-name: String	The name of the person.
-age: int	The age of the person.
-weight: double	The weight of the person in pounds.
-height: double	The height of the person in inches.
+BMI(name: String, age: int, weight: double, height: double)	Creates a BMI object with the specified name, age, weight, and height.
+BMI(name: String, weight: double, height: double)	Creates a BMI object with the specified name, weight, height, and a default age 20.
+getBMI(): double	Returns the BMI
+getStatus(): String	Returns the BMI status (e.g., normal, overweight, etc.)



```
BMI class

- name: String
- age: int
- weight: double
- height: double
+ final KILOGRAMS_PER_POUND: static double
+ final METERS_PER_INCH: static double

<< Constructors >>
+ BMI(String name, int age, double weight, double height)
+ BMI(String name, double weight, double height)
<< Methods >>
+ getBMI(): double
+ getStatus(): String
+ getName(): String
+ getAge() :int
+ getWeight() : double
+ getHeight(): double
+ setName(String name): void
+ setAge(int age): void
+ setWeight(double weight): double
+ setHeight(double height): double
+ equals(BMI b): boolean
+ compareTo(BMI b): double
+ toString(): String
```

The BMI Class Complete Version

getBMI() method is an implicit method (implicit variable, or derived variable). The class stores only **weight** and **height**.

The **BMI** number is calculated on method calls. The class does not store it.

The **Status** string is also calculated on method calls. The class does not store it, neither.

Review Use of Violet UML Tool

- If you did not have violet UML tool, download from here: http://horstmann.com/violet/
- Try to learn how to design UML class (we need other UML knowledge later.)

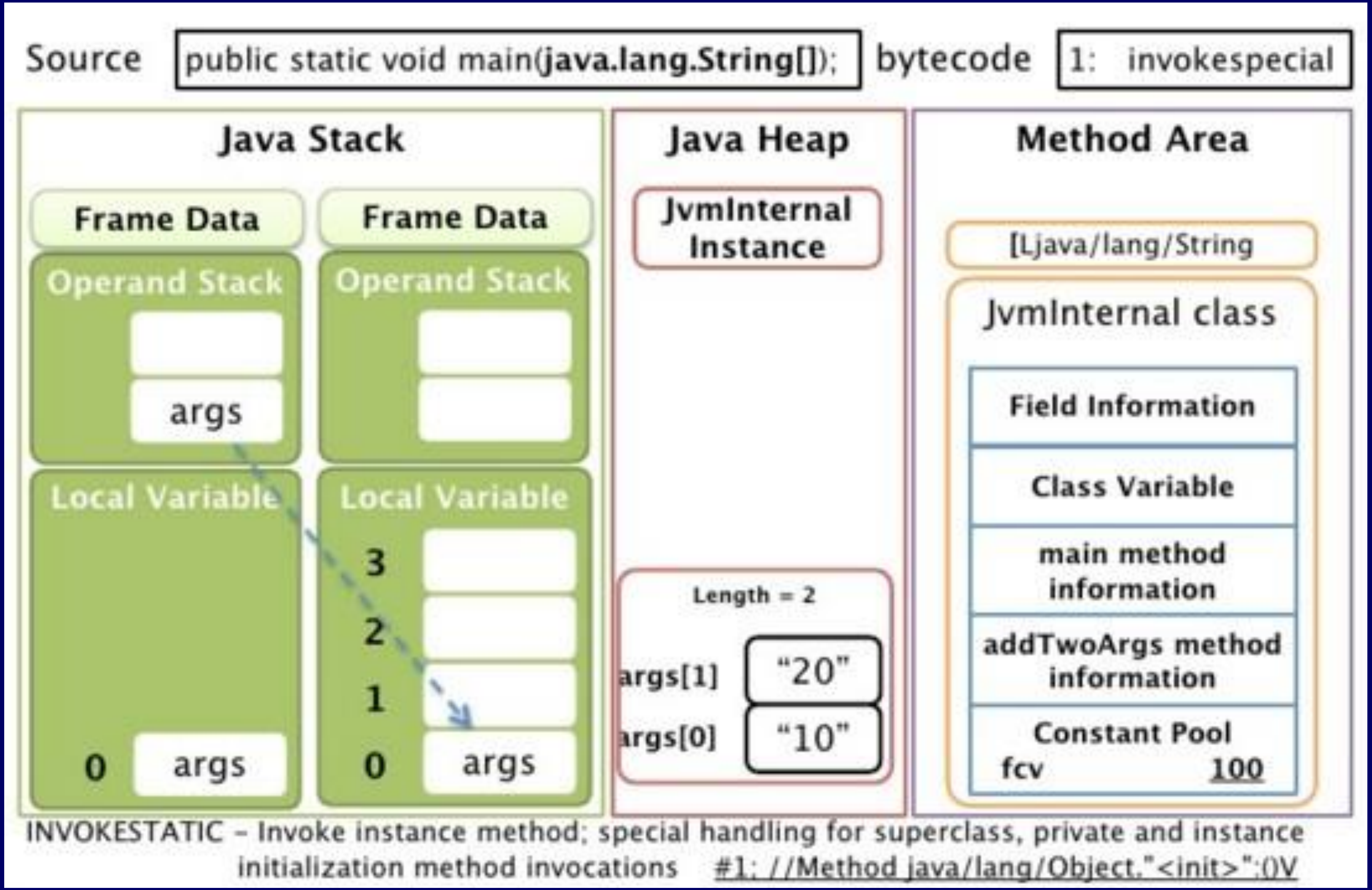
Demonstration Program

BMI/TESTBMI CLASS

Class Loading Time and Data Memory Model

LECTURE 4

Memory Allocation

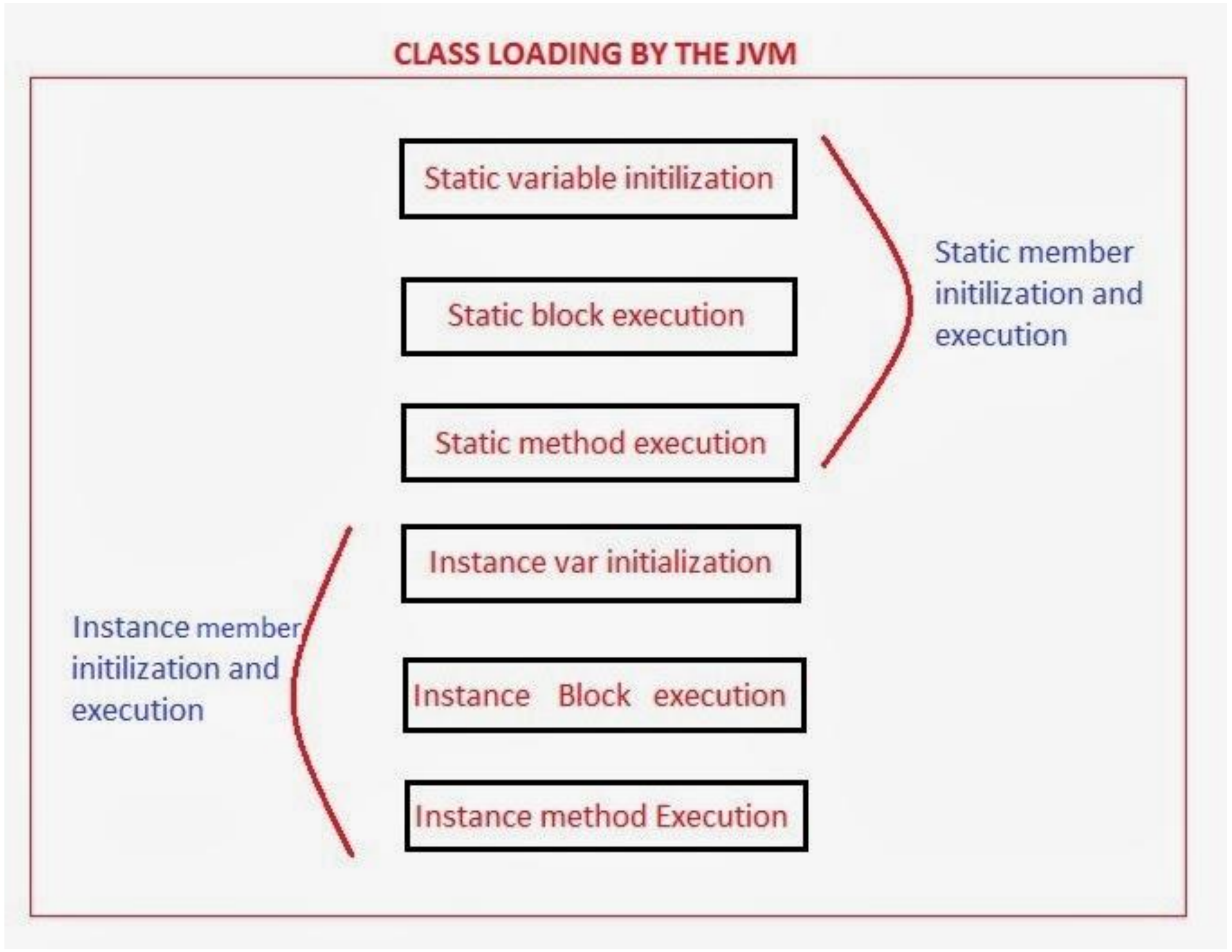


INVOKESTATIC – Invoke instance method; special handling for superclass, private and instance initialization method invocations #1: //Method java/lang/Object."<init>":()V

Loading a Class to JVM

Static Variable/Static Method:

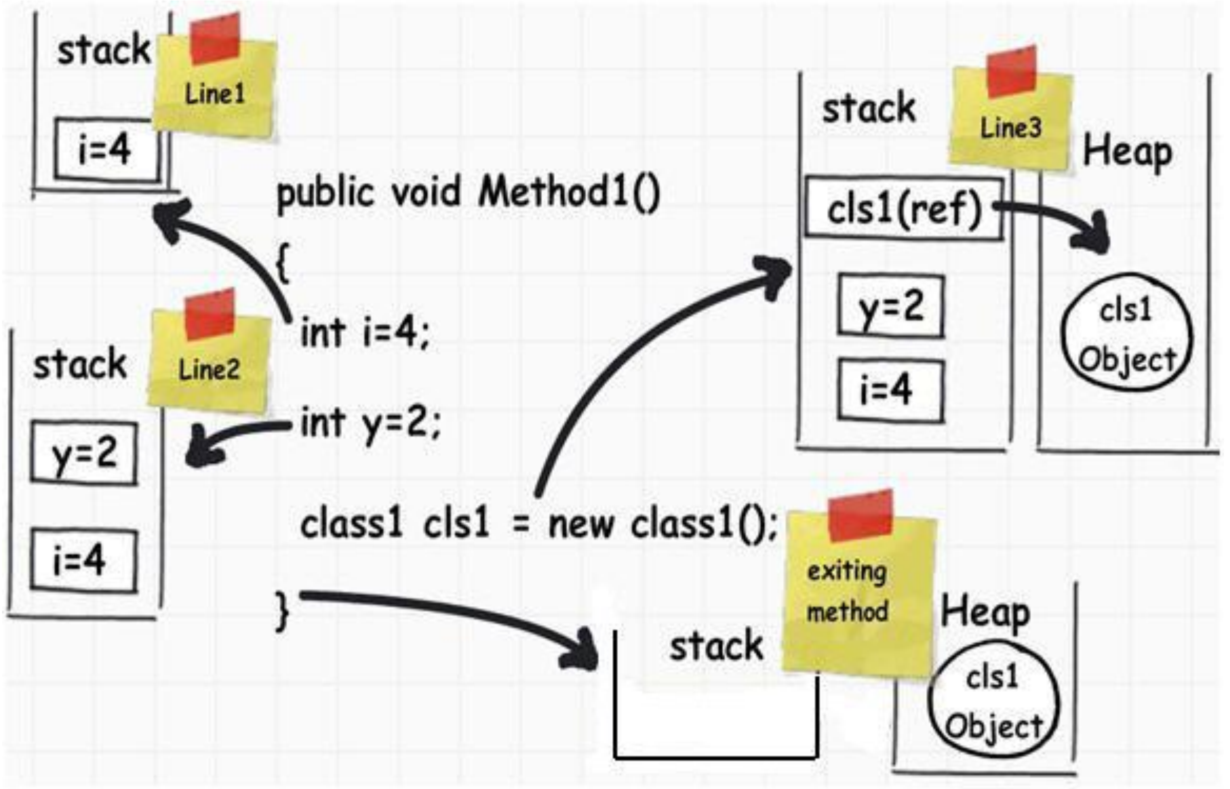
- Constants: (final static variables)
  - Math.PI, Integer.MIN_VALUE
- Class Variable: (static variables)
  - Circle.count
- Utility Method: (static methods)
  - Math.random(), Math.abs(), Integer.parseInt()


CLASS LOADING BY THE JVM
Static variable initilization
Static block execution
Static method execution
Static member initilization and execution
Instance var initialization
Instance Block execution
Instance method Execution
Instance member initilization and execution

Object-Oriented Programming
(scope, visibility modifiers, static modifiers, and final modifier)

Instance Variable/Instance Method:

- encapsulated data fields: (private data)
- un-protected data: (public data)
- accessor/mutator methods: (public method)
- client methods: (private method)

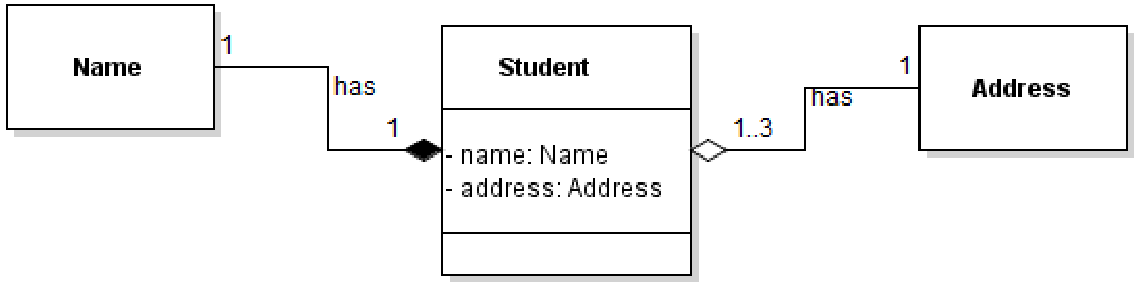


Class has_A Relationship

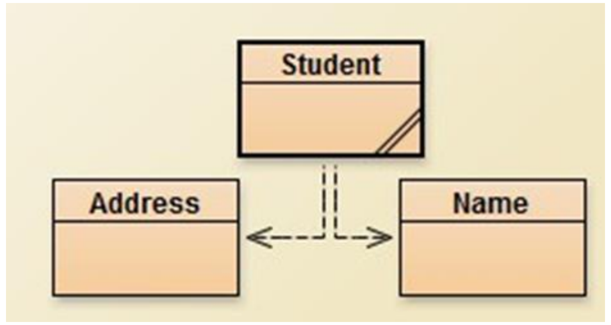
LECTURE 5

has A Relationship

Violet



BlueJ

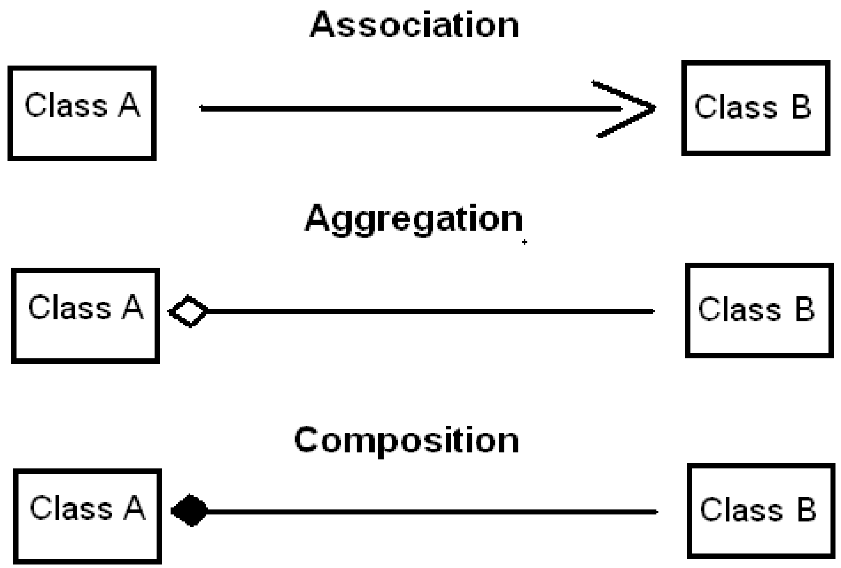


Class Relationships **(use and inherit)**

To design classes, you need to explore the relationships among classes. The common relationships among classes are *association*, *composition*, and *inheritance (later)*

These three relationship is use-relationship in BlueJ.

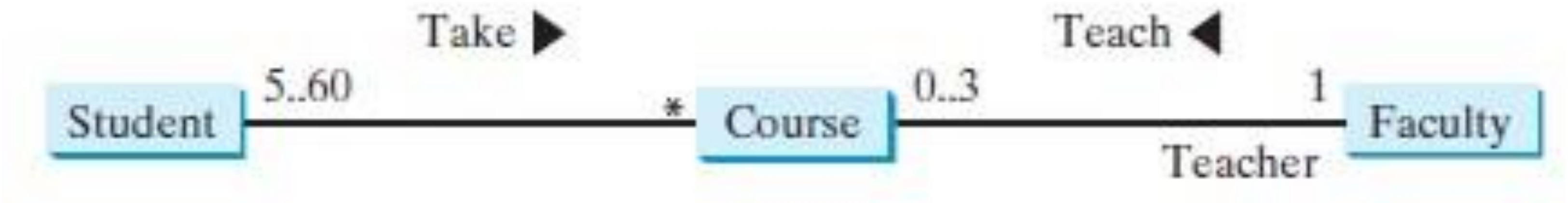
- Association is general case. [many(one)-to-many(one)]
- Aggregation is has-a relationship. (many-to-one/one-to-one)
- Composition is exclusive has-a relation. (one-to-one)



Association

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

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



An association is illustrated by a **solid line** between two classes with an optional label that describes the relationship. (Sometimes an **arrow line** is used.) In Figure above, the labels are ***Take*** and ***Teach***. Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the direction indicates that a student takes a course (as opposed to a course taking a student).

Multiplicity

placed at the side of the class to specify how many of the class's objects are involved in the relationship in UML.

- A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship.
- The character * means an unlimited number of objects, and
- the interval **m..n** indicates that the number of objects is between **m** and **n**, inclusively.
- In Figure of previous page, each student may take any number of courses, and each course must have **at least five and at most sixty** students. Each course is taught by only one faculty member, and a faculty member may teach **from zero to three** courses per semester.

Association Relationship

```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

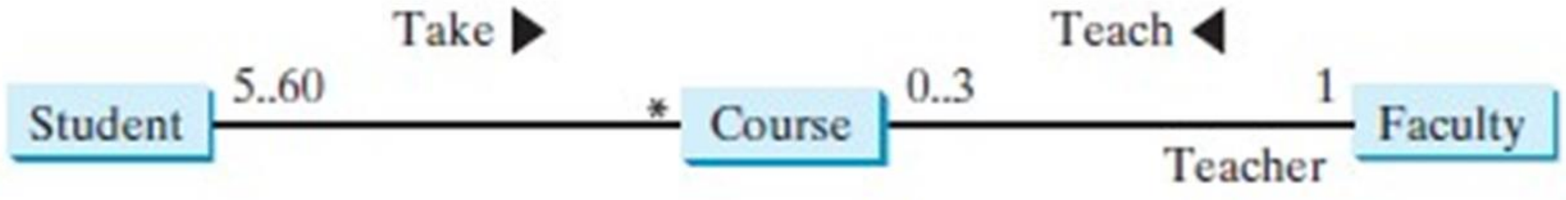
```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

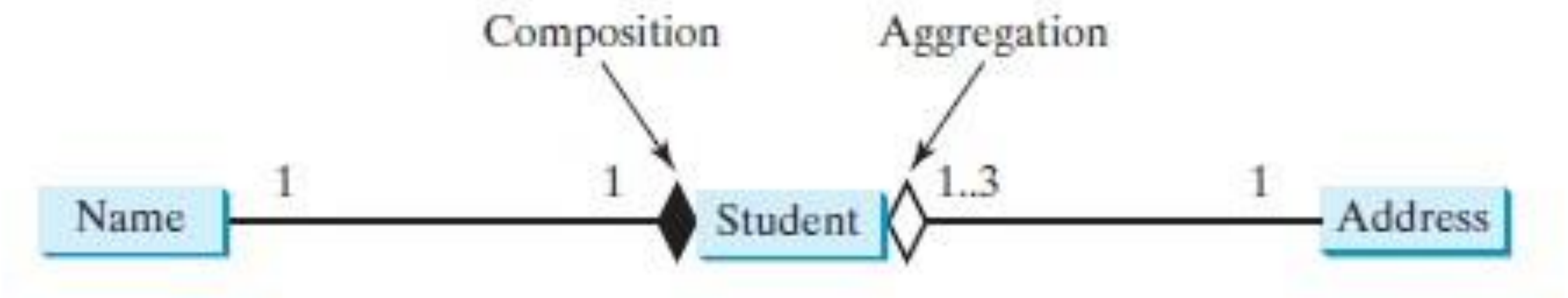
```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```



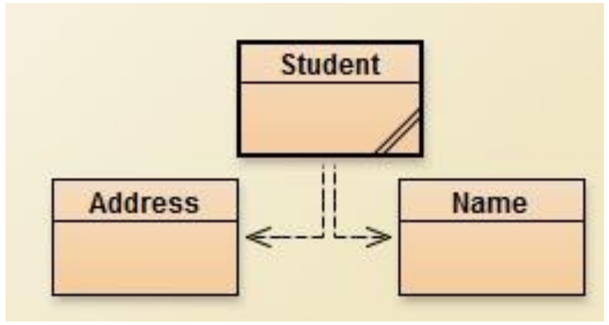
Aggregation or Composition

- Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.



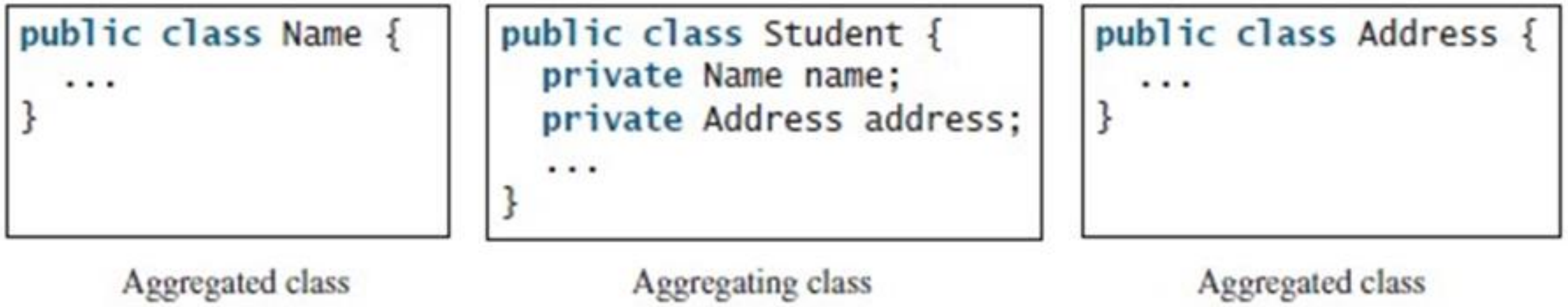
Student has a Name exclusively. (Composition)

Student has an address non-exclusively. (Aggregation)



Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure of previous slide can be represented as follows:



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

Aggregated class

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

Aggregating class

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

Aggregated class

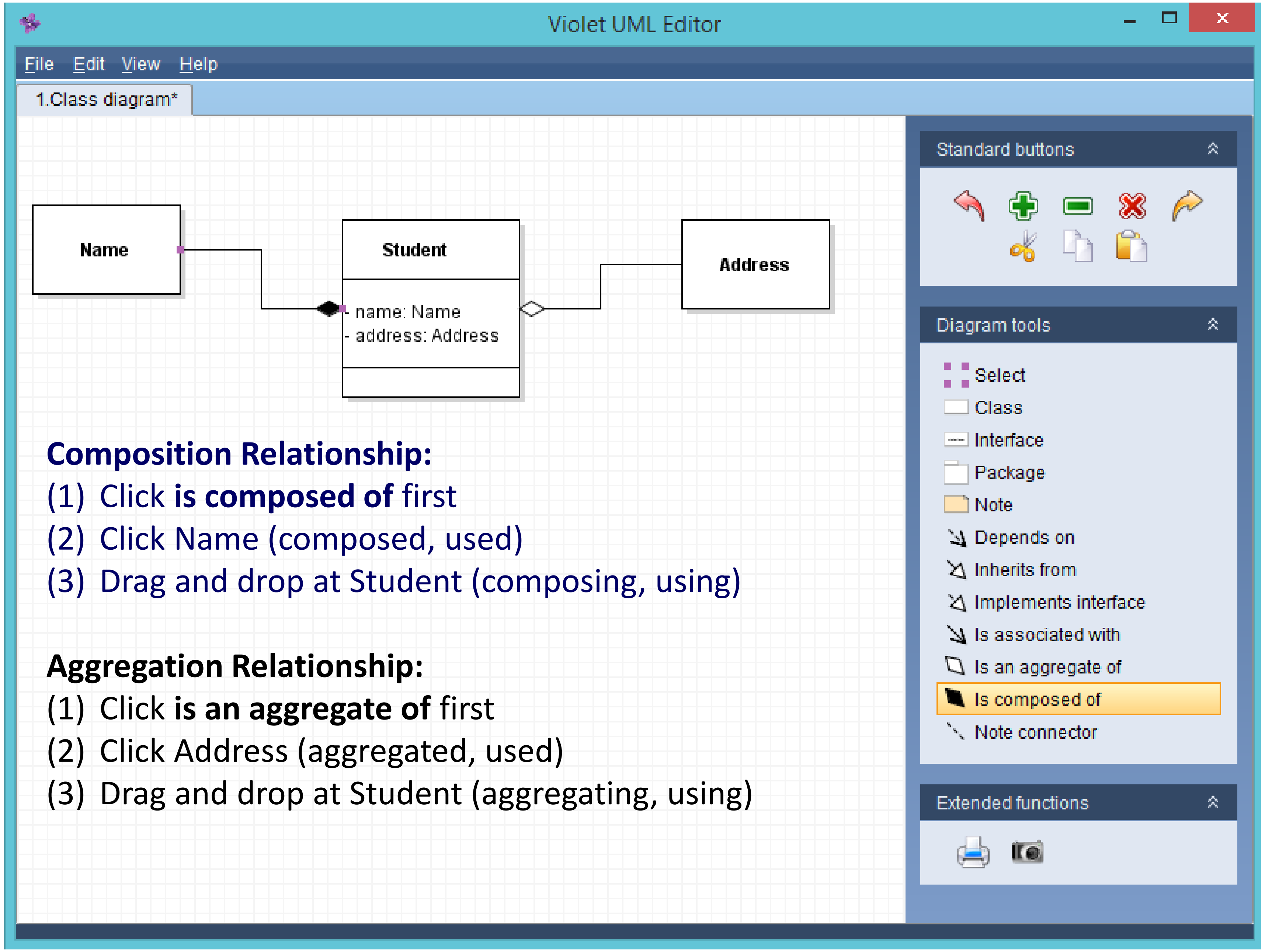
Aggregating: Using   Aggregated: Used

Aggregation (has-a Relationship)

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

- An object can be owned by several other **aggregating objects**.

- If an object is exclusively owned by an aggregating object, the relationship between the object and its aggregating object is referred to as a ***composition***.



- name: Name
- address: Address

Composition Relationship:

(1) Click **is composed of** first

(2) Click Name (composed, used)

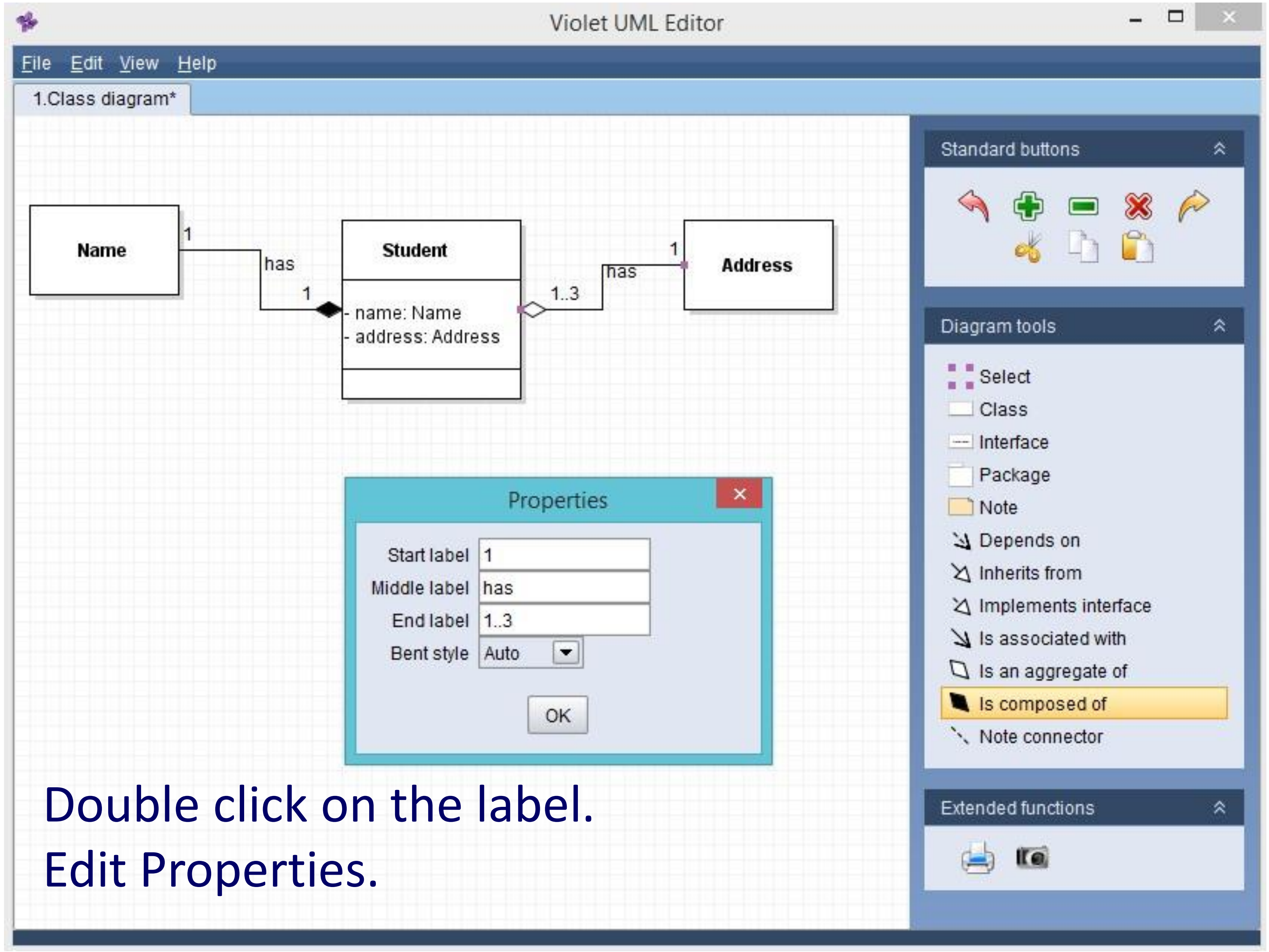
(3) Drag and drop at Student (composing, using)

Aggregation Relationship:

(1) Click **is an aggregate of** first

(2) Click Address (aggregated, used)

(3) Drag and drop at Student (aggregating, using)



Properties

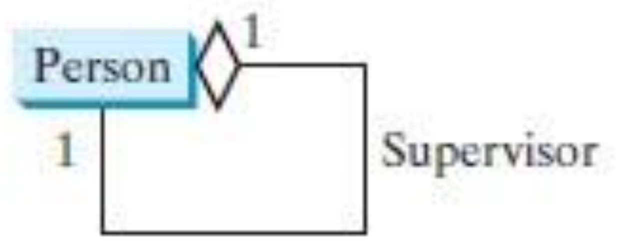
Start label	1
Middle label	has
End label	1..3
Bent style	Auto

Double click on the label.

Edit Properties.

Aggregation Between Same Class

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. **(Using itself once)**

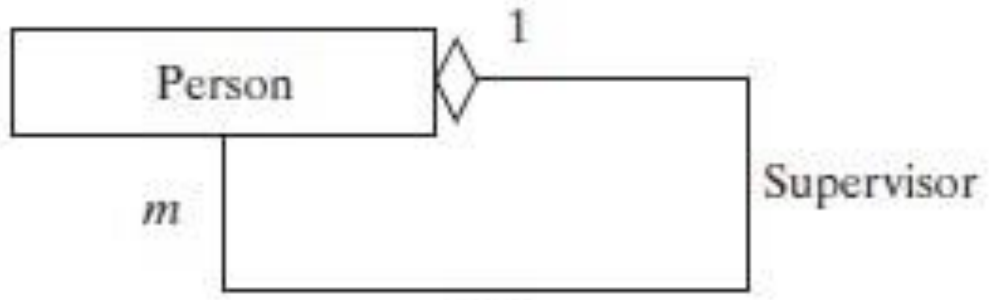


```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;
  ...
}
```

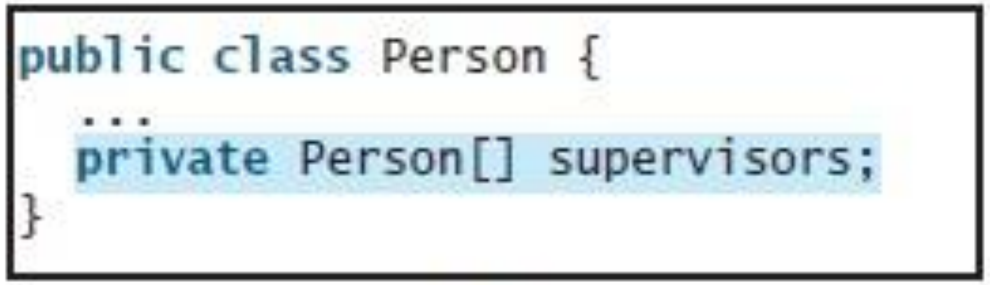
Aggregation Between Same Class

What happens if a person has several supervisors?

(One class using itself many times)



(a)



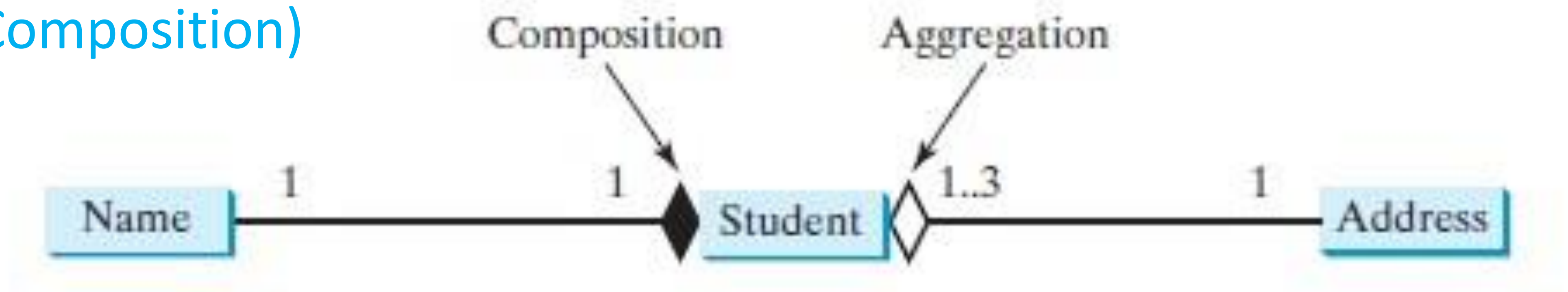
```
public class Person {
  ...
  private Person[] supervisors;
}
```

(b)

Object Composition

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. 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*.

Student has a Name exclusively.
(Composition)

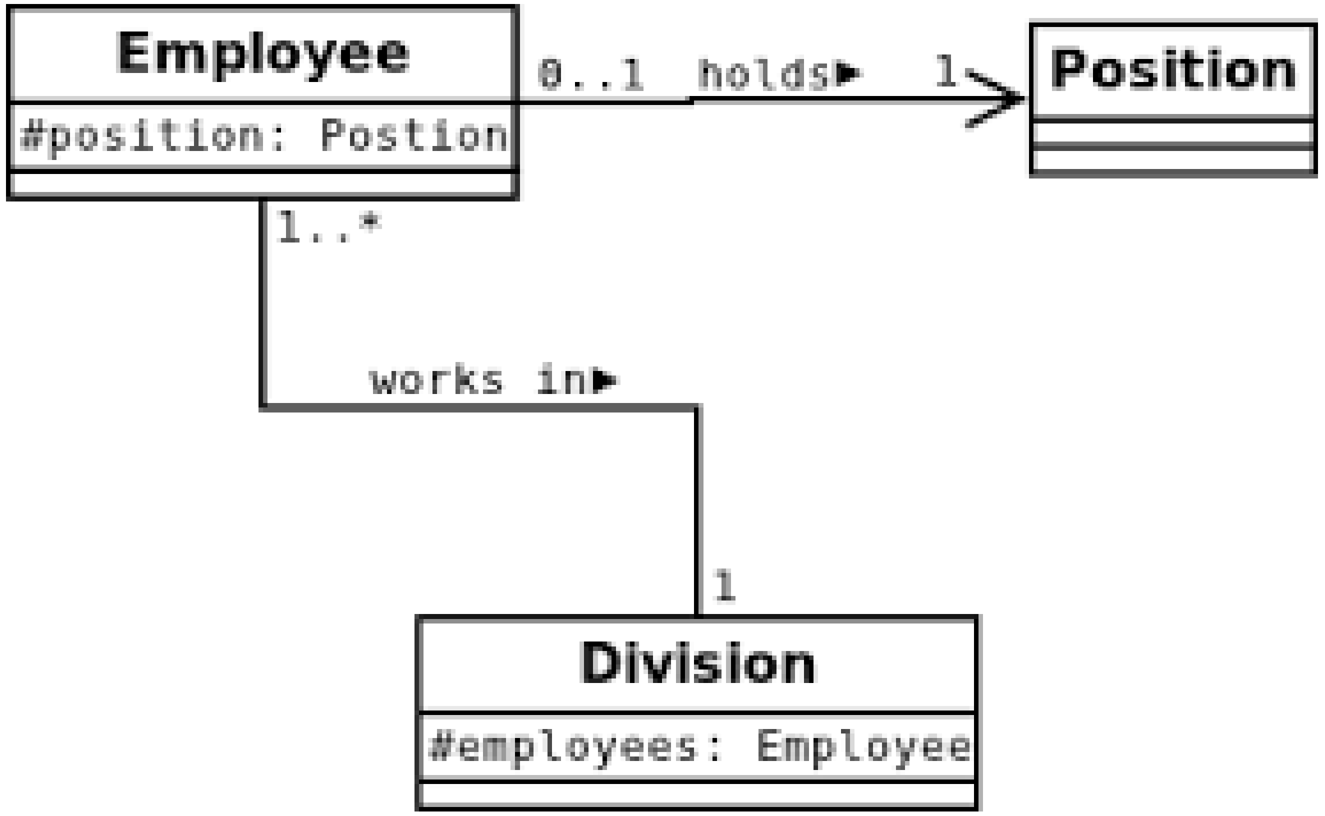


Why analyze the multiplicity of relationship?

http://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561

Widely used in database design. When inner-join, outer-join are to be performed , analysis of these relationship is very essential.

It will be very important for data science.



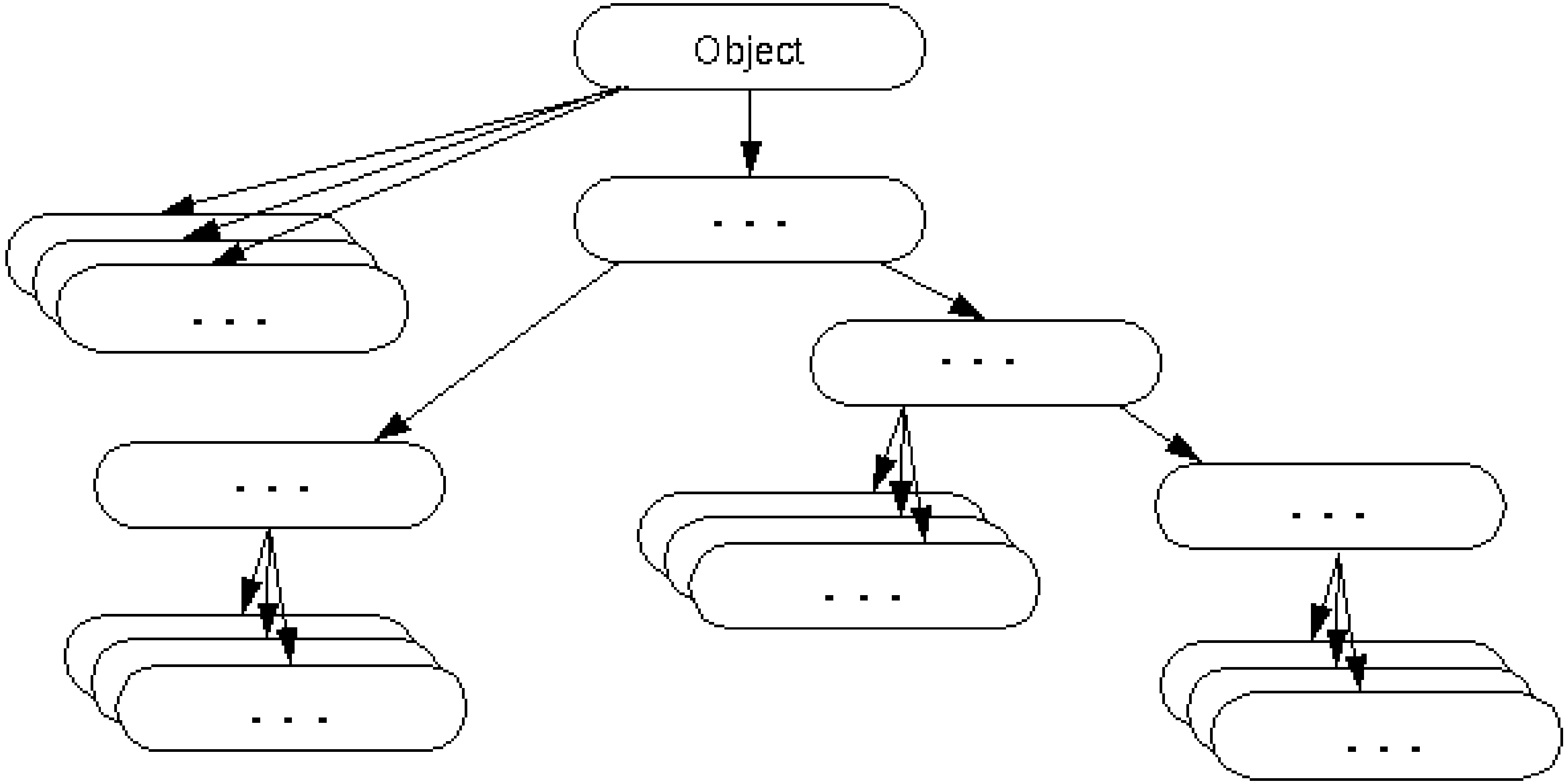
Class Is_A Relationship

LECTURE 6

Class to Class Relationship – Is_A Relationship

- The second type of class-to-class relationship is **Is_A** relationship.
- Before we study the **is_A** relationship, we will look at the Object class and its functions.

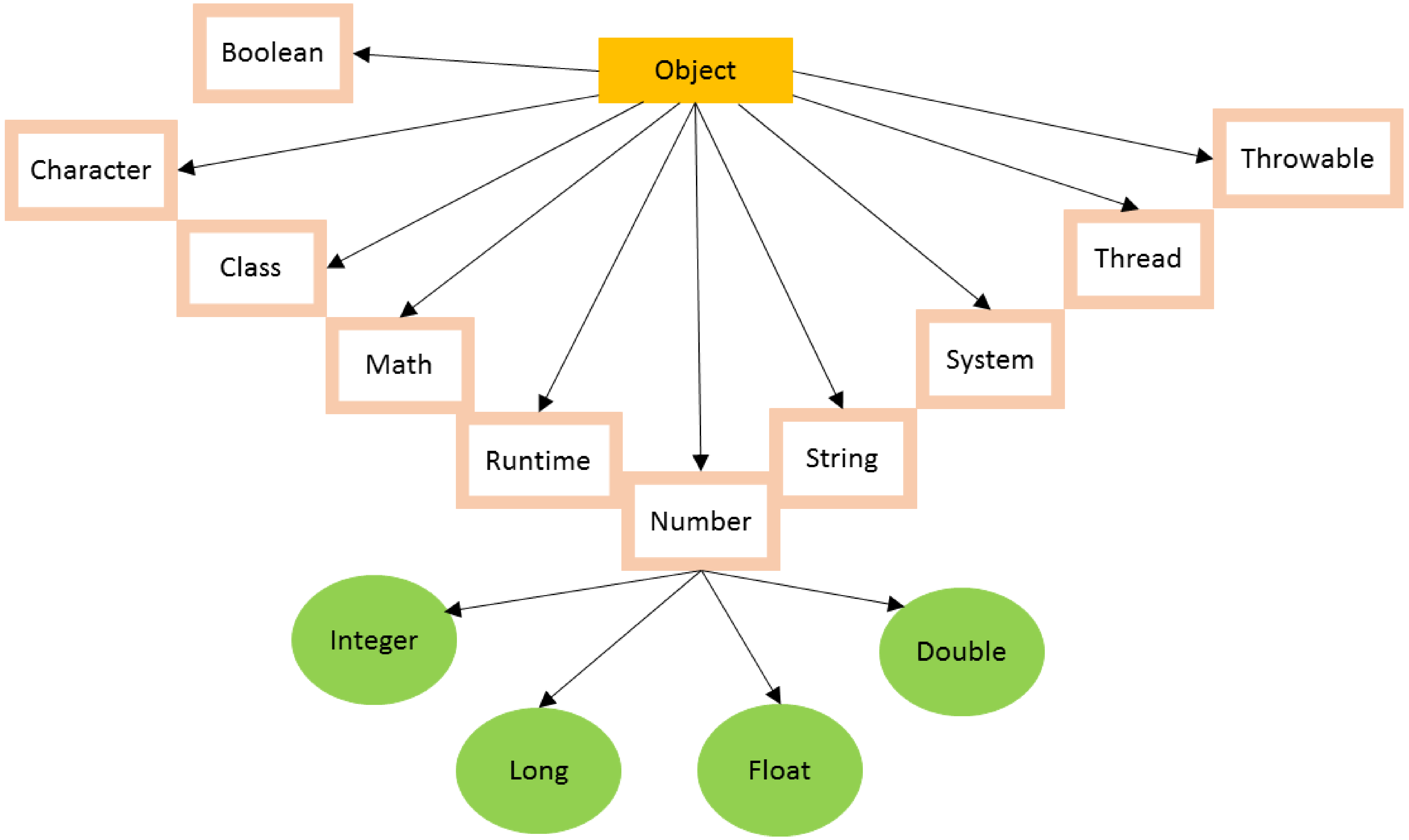
Every Class Inherits from Object Class



Object as a Superclass

Object is superclass for all objects in java.lang language such as Integer, Double, String, Arrays, and etc.

- The Object class, in the **java.lang** package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. **Every class you use or write inherits the instance methods of Object.** You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class.
- The methods inherited from Object that are discussed in this section are:


Object
Boolean
Character
Class
Math
Runtime
Number
String
System
Thread
Throwable
Integer
Long
Float
Double

Object as a Superclass

Object is superclass for all objects in java.lang language such as Integer, Double, String, Arrays, and etc.

- **protected Object clone()** throws CloneNotSupportedException

  Creates and returns a copy of this object.
- **public boolean equals(Object obj)**

  Indicates whether some other object is "equal to" this one.
- **protected void finalize() throws Throwable**

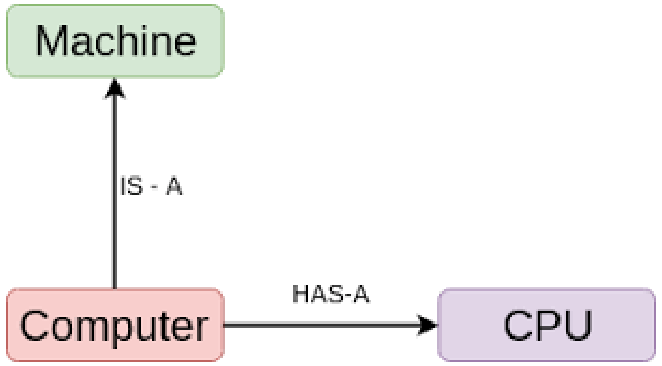
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- **public final Class getClass()**

  Returns the runtime class of an object.
- **public int hashCode()**

  Returns a hash code value for the object.
- **public String toString()**

  Returns a string representation of the object.

Subclass Polymorphism



Generalization (Grouping): Java polymorphism creating a subclass object using its superclass variable.

Overloading: Inheritance of Member Methods (Object Class)

Is_A Relationship: Subclass object is also a superclass object.

Standard Methods for Object Class

LECTURE 7

The clone() Method

One way to copy

- If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a **copy** from an existing object. To create a clone, you write:

```
aCloneableObject.clone();
```

The clone() Method

One way to copy

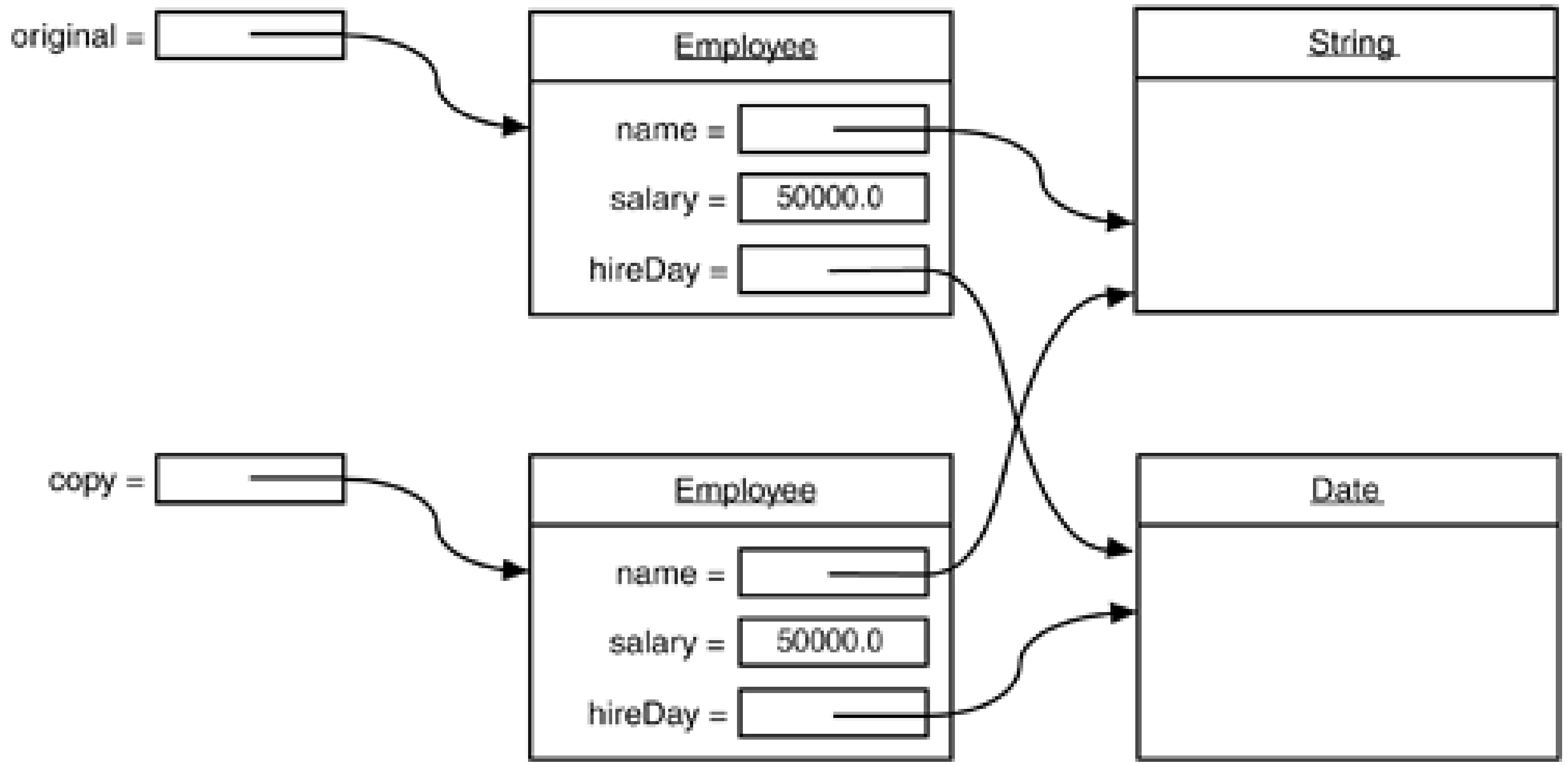
- Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception. Exception handling will be covered in a later lesson. For the moment, you need to know that clone() must be declared as

  ```
  protected Object clone() throws CloneNotSupportedException
  ```

  or:

  ```
  public Object clone() throws CloneNotSupportedException
  ```

clone() method
(Sometimes shallow copy, sometimes deep copy, need to check)



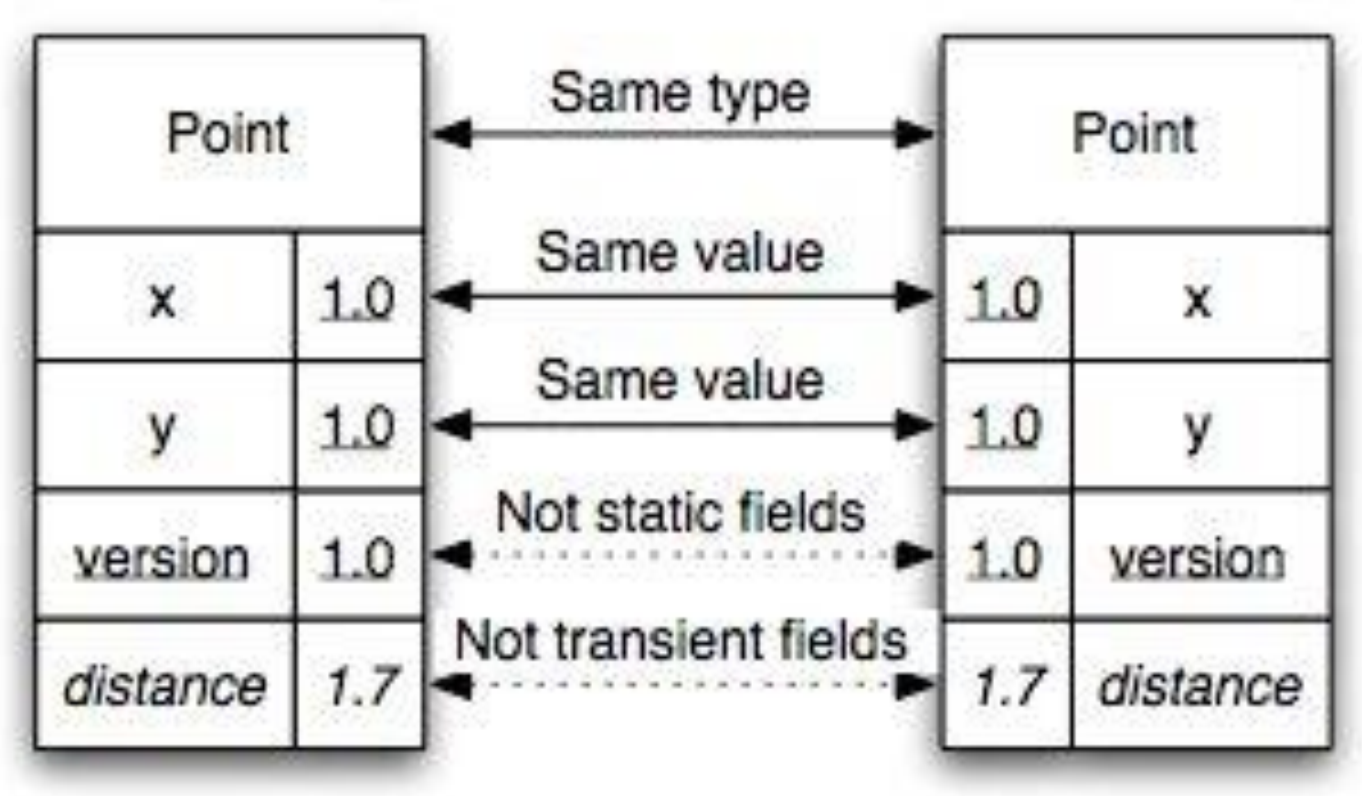
The equals() Method

- The equals() method compares two objects for equality and returns true if they are equal. The **equals()** method provided in the Object class uses the identity operator **(==)** to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The **equals()** method provided by Object tests whether the object references are equal—that is, if the objects compared are the exact same object.
- To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the **equals()** method. Here is an example of a Book class that overrides **equals()**:

The equals() Method

```
public class Book {
    ...
    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book)obj.getISBN());
        else
            return false;
    }
}
```

Equality Check



Static data field is shared.
Nothing to compare.

Overriding equals() gives us a chance to redefine equality

Different definition for equality of Book class:

- A Book is equal if the book’s title is the same.
- A Book is equal if the book’s ISBN is the same. (same print, or same edition)
- A Book is equal if the book is the same copy. (For school library management)

The hashCode() Method

(another equality compare method)

- The value returned by hashCode() is the object's hash code, which is the object's **memory address** in hexadecimal.
- By definition, if two objects are equal, their hash code must also be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

The toString() Method

- You should always consider overriding the **toString()** method in your classes.
- The Object's **toString()** method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override **toString()** in your classes.
- You can use **toString()** along with System.out.println() to display a text representation of an object, such as an instance of Book:

```
System.out.println(firstBook.toString());
```

- which would, for a properly overridden **toString()** method, print something useful, like this:

```
ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition
```

What you need to remember when overriding toString() manually?

- Return as much information as needed (that may be interesting)
- It is obligatory in data classes
- if you decide that your **toString()** provide result in format presentable to the user, then you have to clearly document output print format and remain it unchanged for life. In that case you need to be aware that **toString()** output may be printed in UI somewhere
- beside **toString()** you still need to provide accessor methods for class fields, if needed

The getClass() Method

(Class object is a information object of another object. It is like Color/Font Class)

- You cannot override getClass.
- The **getClass()** method returns a **Class** object, which has methods you can use to get information about the class, such as its name (**getSimpleName()**), its superclass (**getSuperclass()**), and the interfaces it implements (**getInterfaces()**). For example, the following method gets and displays the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The object's" + " class is
" +
        obj.getClass().getSimpleName());
}
```

The getClass() Method

(Class object is a information object of another object. It is like Color/Font Class)

- The **Class** class, in the **java.lang** package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (**isAnnotation()**), an interface (**isInterface()**), or an enumeration (**isEnum()**). You can see what the object's fields are (**getFields()**) or what its methods are (**getMethods()**), and so on.

getClass() and instanceof

object.getClass() return a **Class** object which contains the **Class** information of the object.

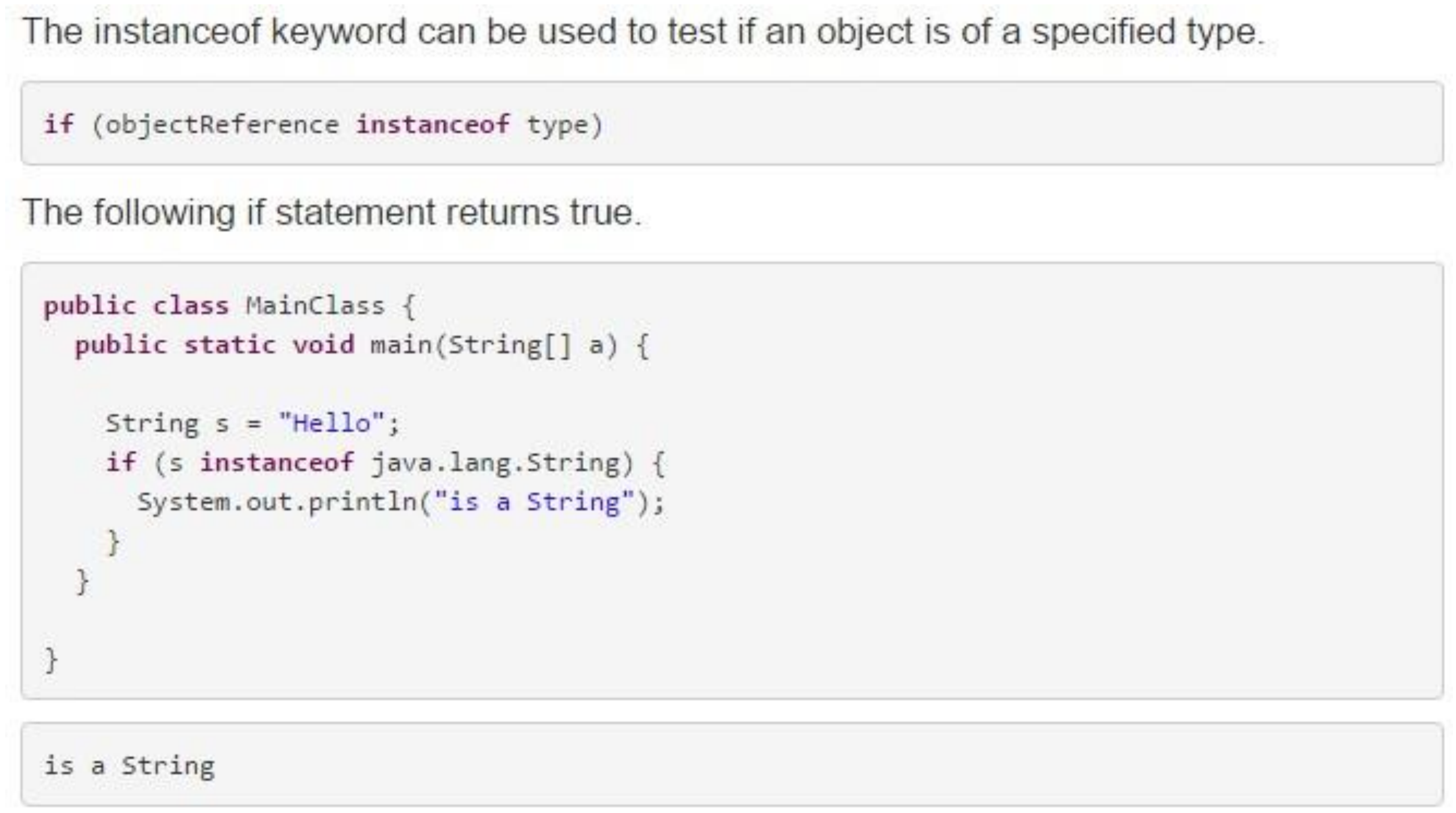
object instanceof class will return a boolean value whether the **object** is of the **class**.

instanceof is an operator (keyword) while **getClass()** is an method.

getClass() has more information than **instanceof**.

Example of instanceof Operator

(if a pointer is null, it will return false)



The instanceof keyword can be used to test if an object is of a specified type.

```
if (objectReference instanceof type)
```

The following if statement returns true.

```
public class MainClass {
  public static void main(String[] a) {

    String s = "Hello";
    if (s instanceof java.lang.String) {
      System.out.println("is a String");
    }
  }

}
```

```
is a String
```

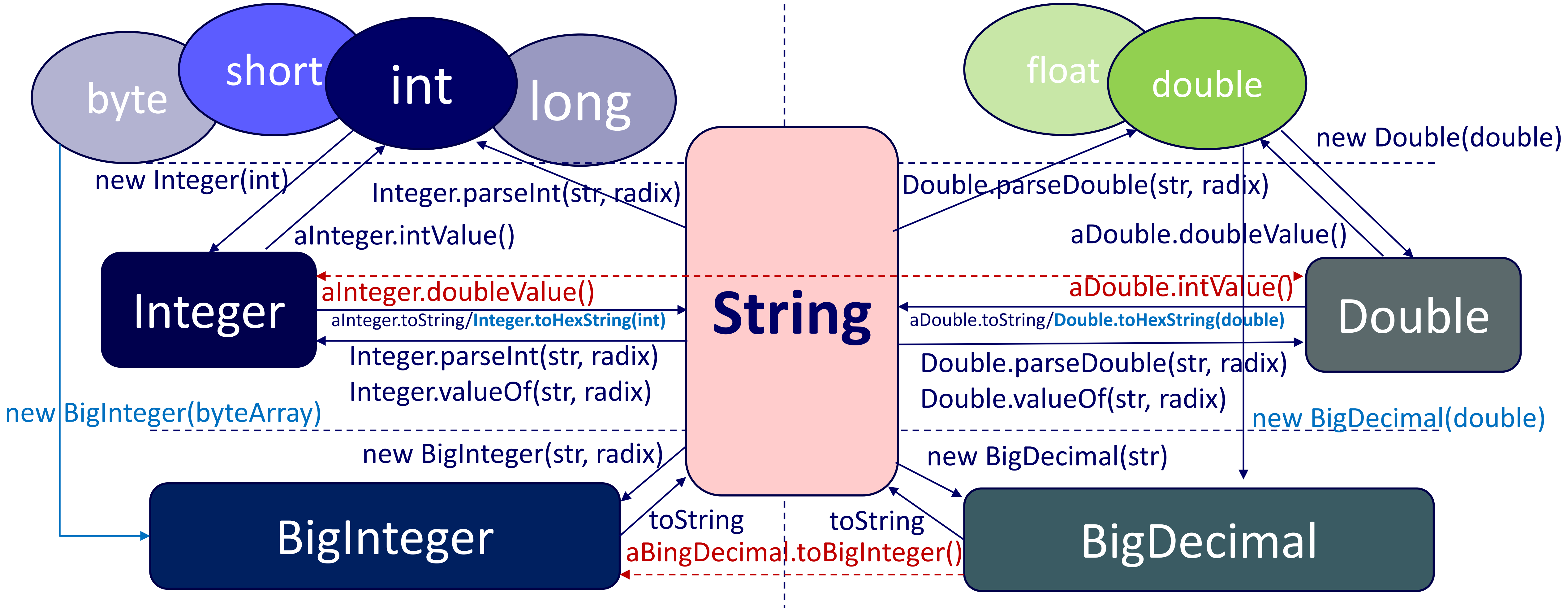
The finalize() Method

- The Object class provides a callback method, **finalize()**, that may be invoked on an object when it becomes garbage. Object's implementation of **finalize()** does nothing—you can override **finalize()** to do cleanup, such as freeing resources.
- The **finalize()** method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect **finalize()** to close them for you, you may run out of file descriptors.

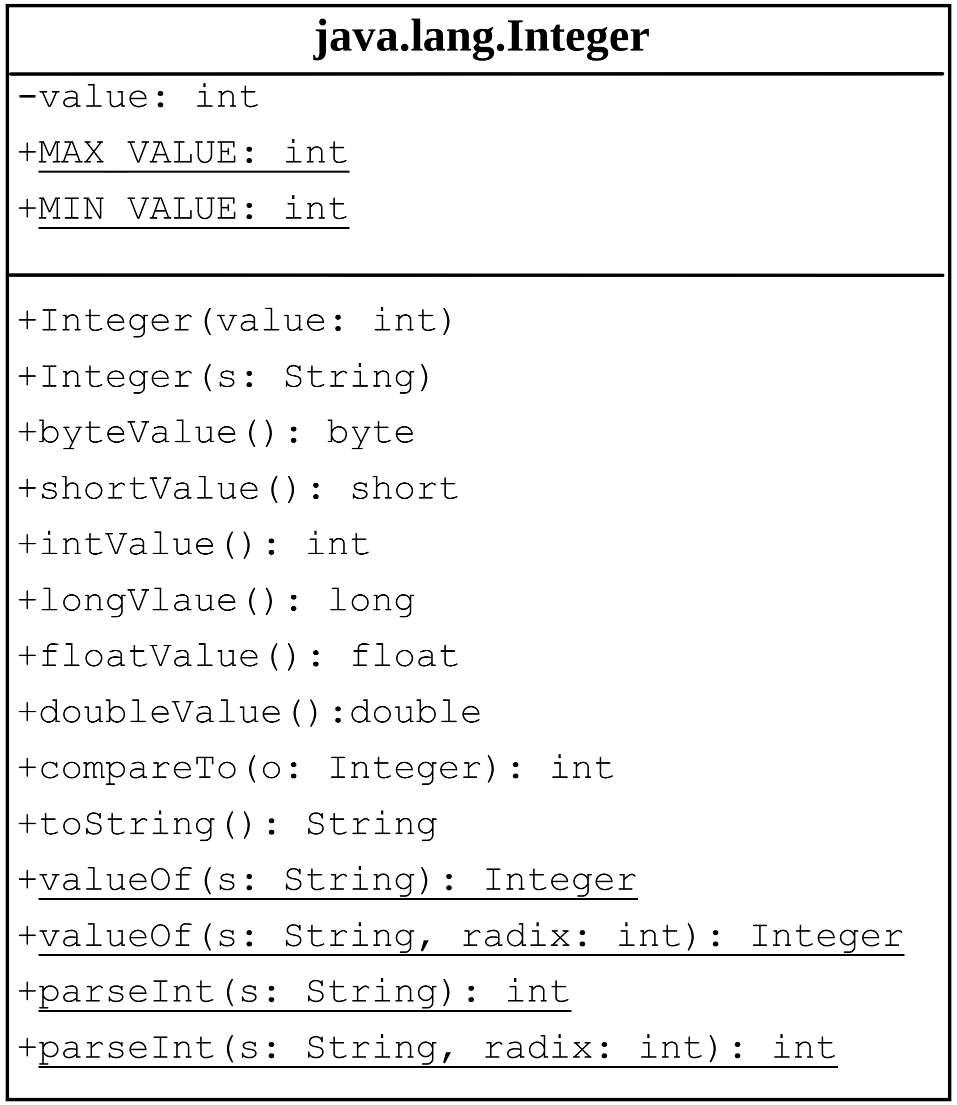
Number Class BigInteger BigDecimal

LECTURE 8

Map for Java Number Space

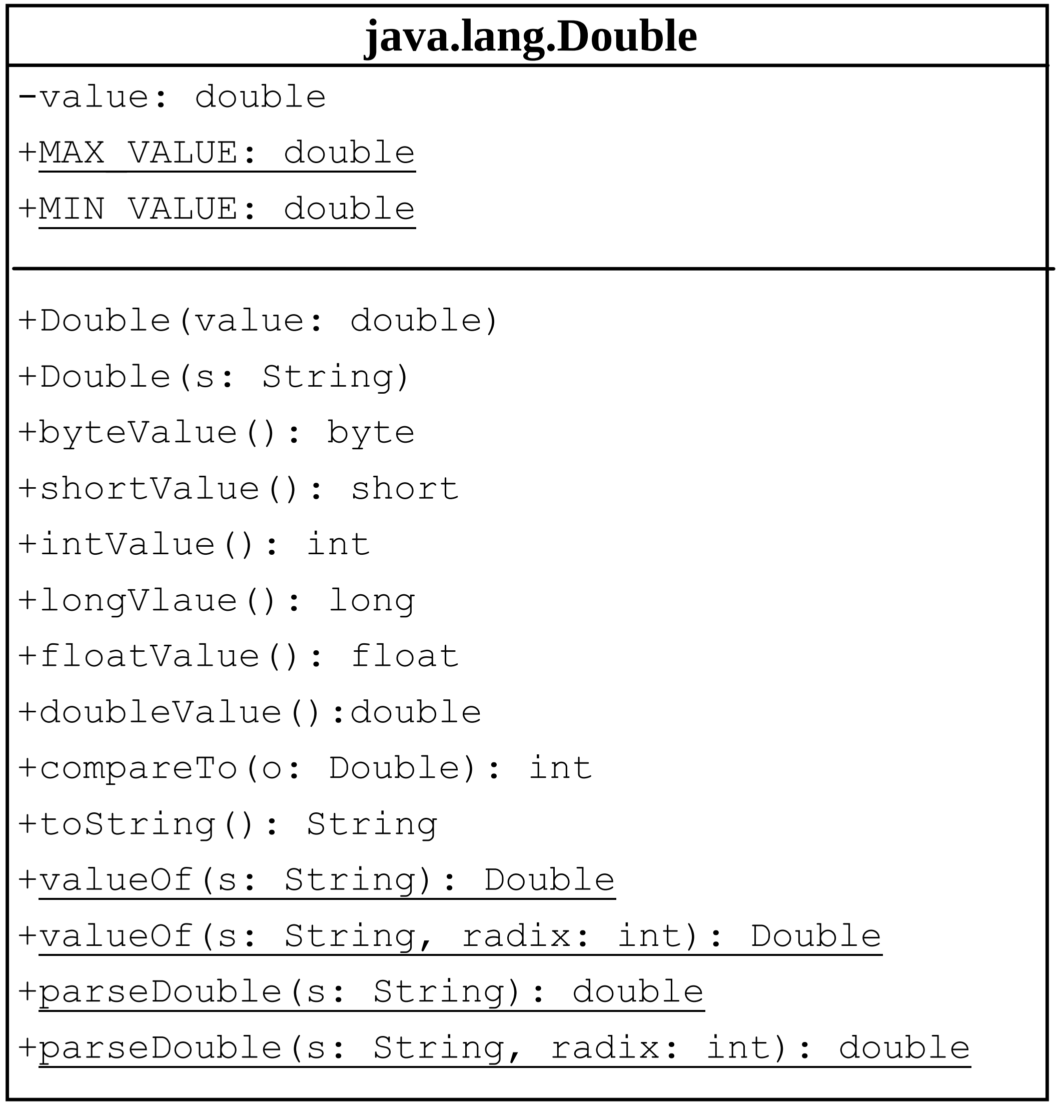

byte
short
int
long
float
double
new Integer(int)
Integer.parseInt(str, radix)
aInteger.intValue()
Double.parseDouble(str, radix)
aDouble.doubleValue()
new Double(double)
Integer
String
Double
aInteger.doubleValue()
aInteger.toString/Integer.toHexString(int)
Integer.parseInt(str, radix)
Integer.valueOf(str, radix)
aDouble.intValue()
aDouble.toString/Double.toHexString(double)
Double.parseDouble(str, radix)
Double.valueOf(str, radix)
new BigInteger(byteArray)
new BigDecimal(double)
new BigInteger(str, radix)
new BigDecimal(str)
BigInteger
toString
toString
aBingDecimal.toBigInteger()
BigDecimal

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
```

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

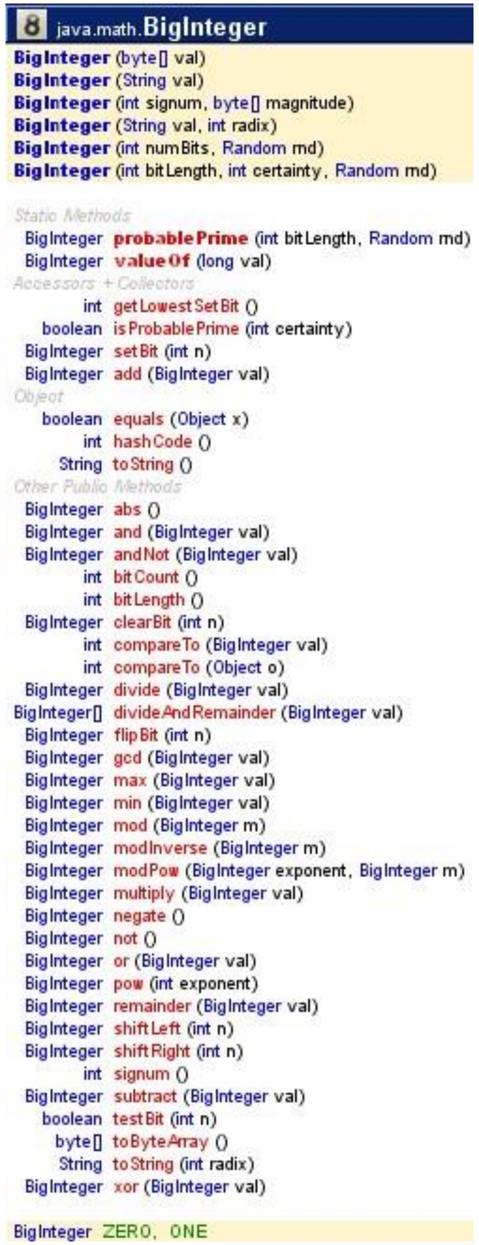
```
Integer[] intArray = {2, 4, 3};
```

(b)

New JDK 1.5 boxing

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

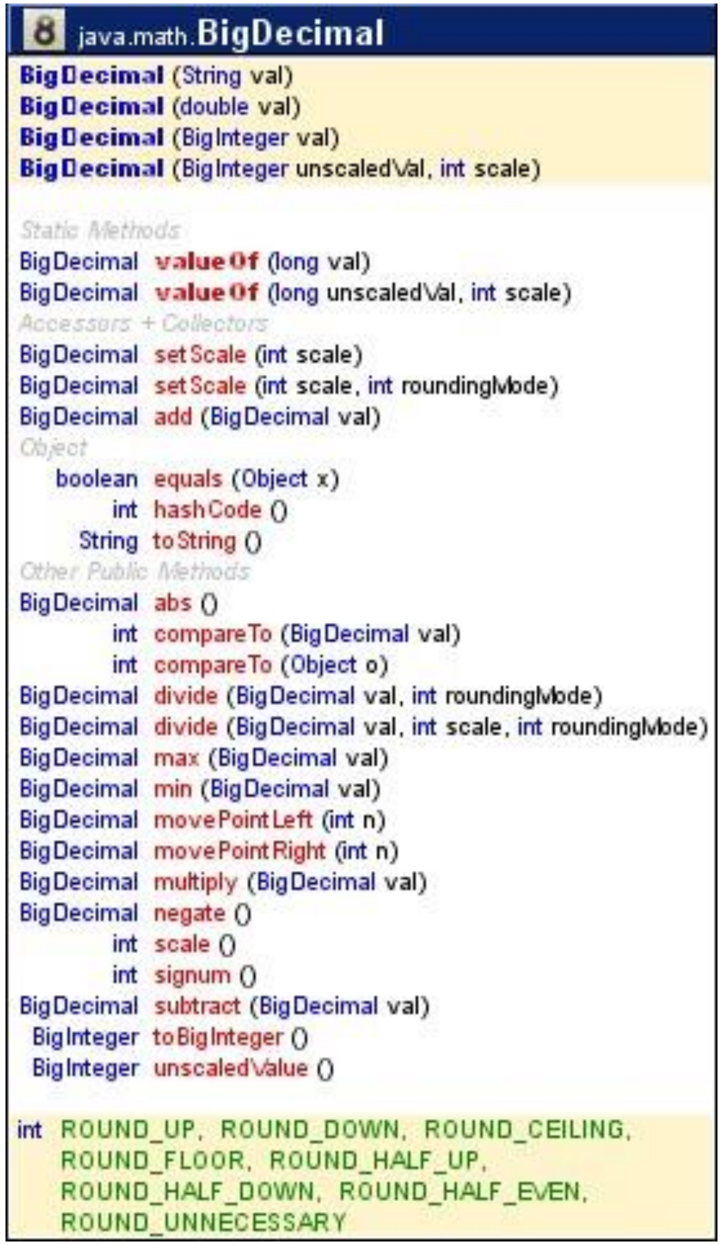
Unboxing



```
8 java.math.BigInteger
BigInteger (byte[] val)
BigInteger (String val)
BigInteger (int signum, byte[] magnitude)
BigInteger (String val, int radix)
BigInteger (int numBits, Random rnd)
BigInteger (int bitLength, int certainty, Random rnd)

Static Methods
  BigInteger  probablePrime (int bitLength, Random rnd)
  BigInteger  valueOf (long val)
Accessors + Collectors
         int  getLowestSetBit ()
     boolean  isProbablePrime (int certainty)
  BigInteger  setBit (int n)
  BigInteger  add (BigInteger val)
Object
     boolean  equals (Object x)
         int  hashCode ()
      String  toString ()
Other Public Methods
  BigInteger  abs ()
  BigInteger  and (BigInteger val)
  BigInteger  andNot (BigInteger val)
         int  bitCount ()
         int  bitLength ()
  BigInteger  clearBit (int n)
         int  compareTo (BigInteger val)
         int  compareTo (Object o)
  BigInteger  divide (BigInteger val)
BigInteger[]  divideAndRemainder (BigInteger val)
  BigInteger  flipBit (int n)
  BigInteger  gcd (BigInteger val)
  BigInteger  max (BigInteger val)
  BigInteger  min (BigInteger val)
  BigInteger  mod (BigInteger m)
  BigInteger  modInverse (BigInteger m)
  BigInteger  modPow (BigInteger exponent, BigInteger m)
  BigInteger  multiply (BigInteger val)
  BigInteger  negate ()
  BigInteger  not ()
  BigInteger  or (BigInteger val)
  BigInteger  pow (int exponent)
  BigInteger  remainder (BigInteger val)
  BigInteger  shiftLeft (int n)
  BigInteger  shiftRight (int n)
         int  signum ()
  BigInteger  subtract (BigInteger val)
     boolean  testBit (int n)
      byte[]  toByteArray ()
      String  toString (int radix)
  BigInteger  xor (BigInteger val)

BigInteger ZERO, ONE
```



```
8 java.math.BigDecimal
BigDecimal (String val)
BigDecimal (double val)
BigDecimal (BigInteger val)
BigDecimal (BigInteger unscaledVal, int scale)

Static Methods
BigDecimal  valueOf (long val)
BigDecimal  valueOf (long unscaledVal, int scale)
Accessors + Collectors
BigDecimal  setScale (int scale)
BigDecimal  setScale (int scale, int roundingMode)
BigDecimal  add (BigDecimal val)
Object
   boolean  equals (Object x)
       int  hashCode ()
    String  toString ()
Other Public Methods
BigDecimal  abs ()
       int  compareTo (BigDecimal val)
       int  compareTo (Object o)
BigDecimal  divide (BigDecimal val, int roundingMode)
BigDecimal  divide (BigDecimal val, int scale, int roundingMode)
BigDecimal  max (BigDecimal val)
BigDecimal  min (BigDecimal val)
BigDecimal  movePointLeft (int n)
BigDecimal  movePointRight (int n)
BigDecimal  multiply (BigDecimal val)
BigDecimal  negate ()
       int  scale ()
       int  signum ()
BigDecimal  subtract (BigDecimal val)
 BigInteger  toBigInteger ()
 BigInteger  unscaledValue ()

int  ROUND_UP, ROUND_DOWN, ROUND_CEILING,
     ROUND_FLOOR, ROUND_HALF_UP,
     ROUND_HALF_DOWN, ROUND_HALF_EVEN,
     ROUND_UNNECESSARY
```

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

(Data Class with Operations)

```
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);
```

In-class Programming

FIBONACCI NUMBERS USING BIGINTEGER

In-class Programming

GOLDEN RATIO USING BIGDECIMAL

StringList

LECTURE 9

Lab Problem Statement

StringList in has_A format

- If there is a data class named StringList, it is designed to store and manipulate a list of names for fruits. This incomplete class declaration is shown below You are required to implement the constructor, two other methods in this class and its testing program.

Lab Problem Statement

StringList in has_A format

```
public class StringList
{
     private ArrayList mList;
     StringList(ArrayList<String> wlist){
          /*put your implementation here. */
     }
     public int numWordsOfLength(int len){ /* put your implementation here. */}

     public void removeWordsOfLength(int len){ /* put your implementation here*/}
}
```

Part (1):

Write the constructor StringList() and numWordsOfLength(int len) method. Method numWordsOfLength returns the number of words in the WordList that are exactly len letters long. For example, assume that the instance variable mList of the StringList fruits contains the following.

```
["lemon‟, "date‟, "mango‟, "kiwi‟, "apple‟, "watermelon‟]
```

The table below shows several sample calls to numWordsOfLength.

Call	Result
fruits.numWordsOfLength(5)	3
fruits.numWordsOfLength(4)	2
fruits.numWordsOfLength(3)	0

Part (2):

Write the StringList method removeWordsOfLength. Method removeWordsOfLength removes all words from the StringList that are exactly len letters long, leaving the order of the remaining words unchanged. For example, assume that the instance variable mList of the String fruits contains the following:

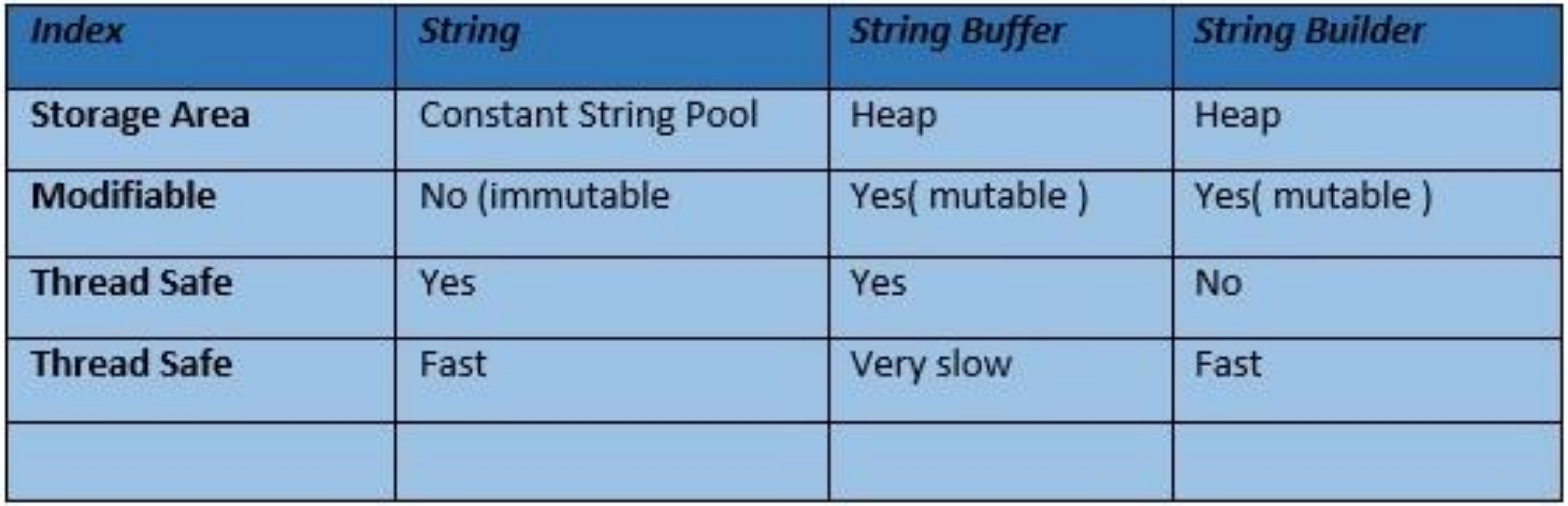
["lemon“, "date“, "mango“, "kiwi“, "apple“, "watermelon“]

Call	Result
fruits.removeWordsOfLength(5)	[date, kiwi, watermelon]
fruits.removeWordsOfLength(4)	[watermelon]
fruits.removeWordsOfLength(3)	[watermelon]

String, StringBuilder, StringBuffer

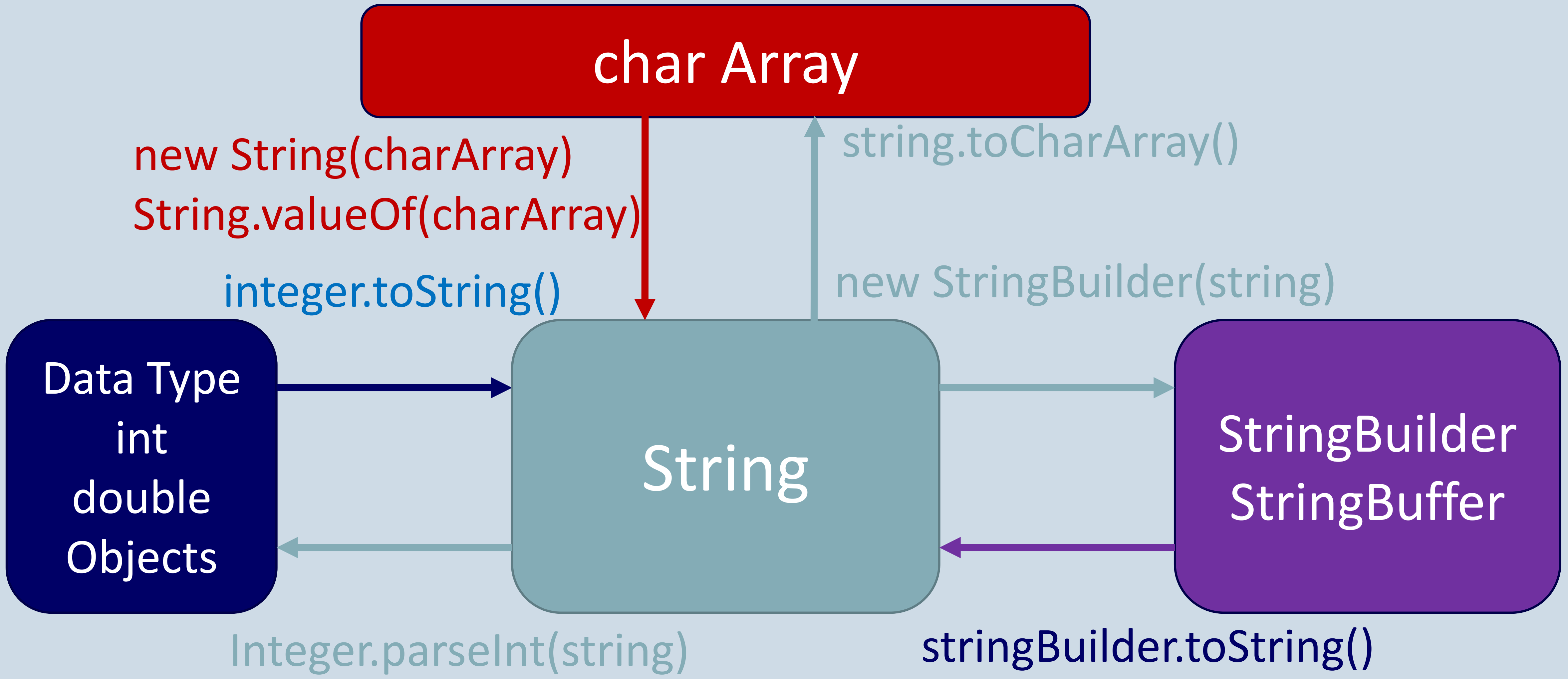
LECTURE 10

String <-> StringBuilder (StringBuffer)



Index	*String*	*String Buffer*	*String Builder*
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable	Yes(mutable)	Yes(mutable)
Thread Safe	Yes	Yes	No
Thread Safe	Fast	Very slow	Fast

String Type Conversion



Review of String Class

Converting, Replacing, and Splitting Strings (check **StringBuilder.java**)

java.lang.String	
+toLowerCase(): String	Returns a new string with all characters converted to lowercase.
+toUpperCase(): String	Returns a new string with all characters converted to uppercase.
+trim(): String	Returns a new string with blank characters trimmed on both sides.
+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".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".replace("e", "AB") returns a new string, WABlcomAB.

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

Splitting a String

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

displays

Java HTML Perl

Finding a Character or a Substring in a String

java.lang.String	
+indexOf(ch: char): int	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
+indexOf(ch: char, fromIndex: int): int	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
+indexOf(s: String): int	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
+indexOf(s: String, fromIndex: int): int	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
+lastIndexOf(ch: int): int	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
+lastIndexOf(ch: int, fromIndex: int): int	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
+lastIndexOf(s: String): int	Returns the index of the last occurrence of string s. Returns -1 if not matched.
+lastIndexOf(s: String, fromIndex: int): int	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

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.

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’.

Conversion between Strings and Arrays

Strings are not arrays, but a string can be converted into an array, and vice versa. To convert a string into an array of characters, use the toCharArray method. For example, the following statement converts the string Java to an array.

```
    char[] chars = "Java".toCharArray();
```

Thus, chars[0] is J, chars[1] is a, chars[2] is v, and chars[3] is a.

You can also use the

getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) method to copy a substring of the string from index srcBegin to index srcEnd-1 into a character array **dst** starting from index dstBegin.

Conversion between Strings and Arrays

For example, the following code copies a substring **"3720"** in **"CS3720"** from index 2 to index 6-1 into the character array dst starting from index 4.

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

Thus, dst becomes {'J', 'A', 'V', 'A', '3', '7', '2', '0'}.

To convert an array of characters into a string, use the String(char[]) constructor or the valueOf(char[]) method. For example, the following statement constructs a string from an array using the String constructor.

```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

The next statement constructs a string from an array using the valueOf method.

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

Formatted Strings

String.format(): create a string, printf(): print directly

The String class contains the static format method to return a formatted string. The syntax
to invoke this method is:

String.format(format, item1, item2, ..., *itemk*)

This method is similar to the **printf** method except that the **format** method returns a formatted
string, whereas the **printf** method displays a formatted string. For example,

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

displays

```
♠♠45.56♠♠♠♠14AB♠♠
```

Formatted Strings

String.format(): create a string, printf(): print directly

Note that

```
    System.out.printf(format, item1, item2, ..., itemk);
```

is equivalent to

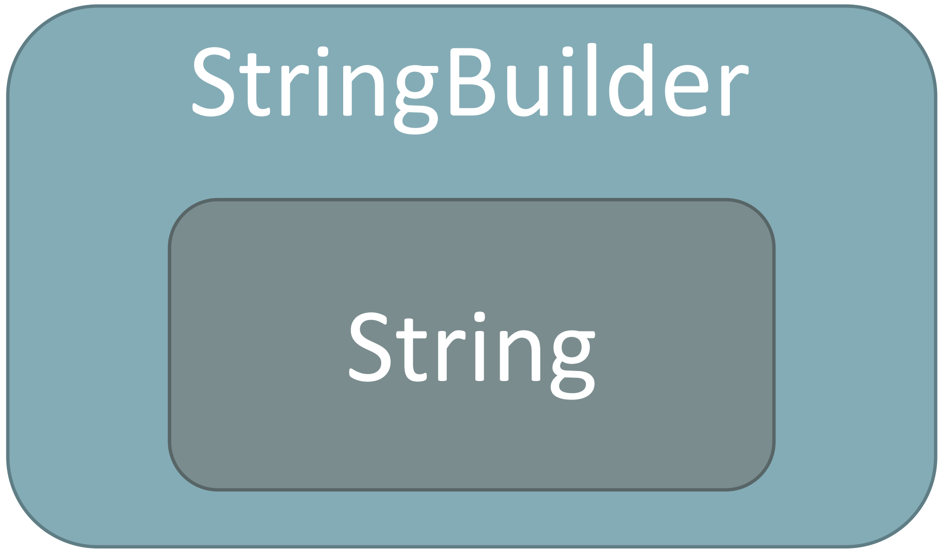
```
    System.out.print(String.format(format, item1, item2, ..., itemk));
```

where the square box (♠) denotes a blank space.

String, StringBuilder, and StringBuffer II

LECTURE 11

StringBuilder is kind of a Wrapper Class for String class (no-auto-boxing unboxing)



Mutators:

append(): like add in ArrayList
insert(): like add in ArrayList
reverse()
delete(): like remove in ArrayList
setCharAt(index, char): like set in ArrayList

Accessors: (like String class)

charAt(index): like get in ArrayList

`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.

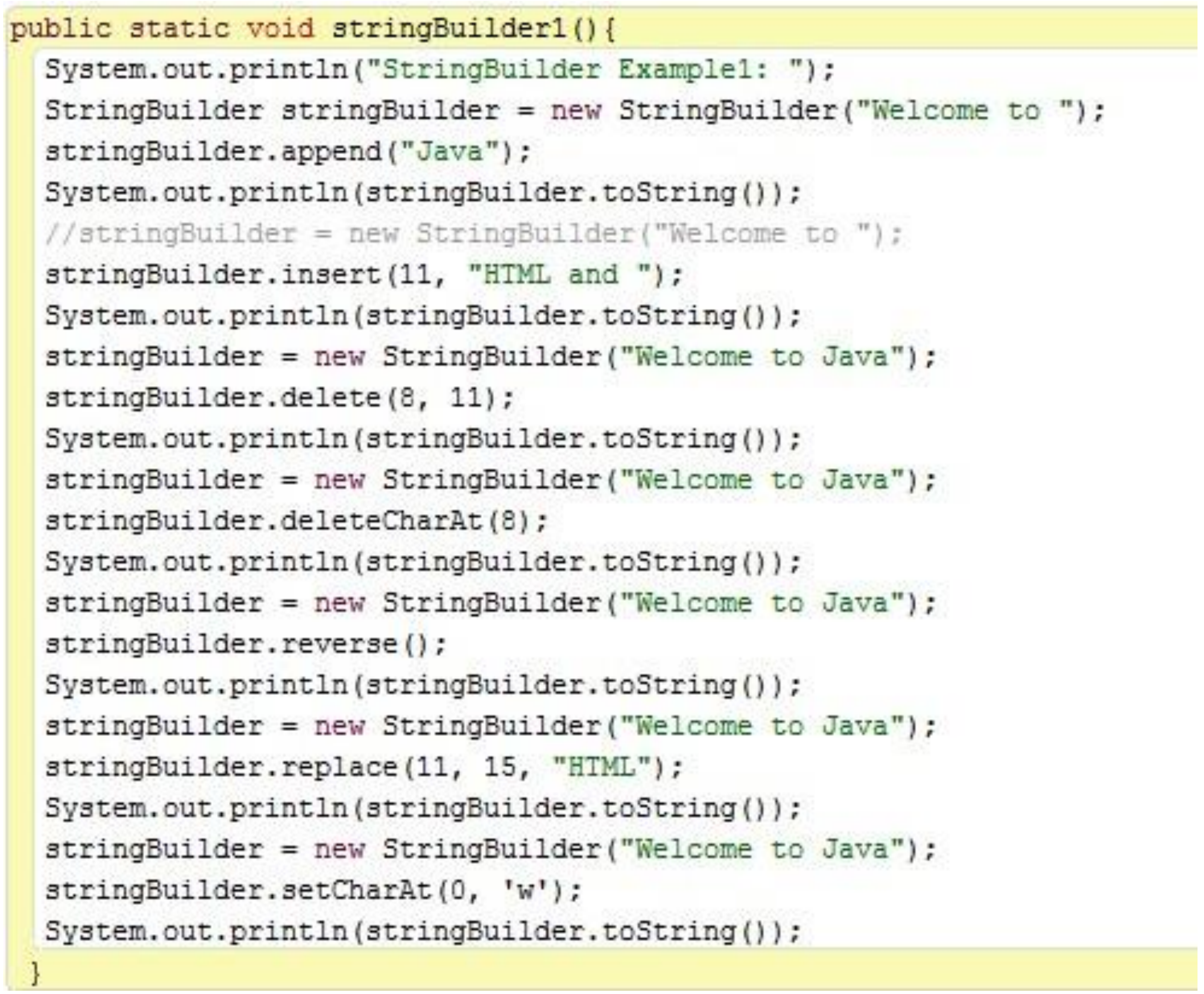
StringBuilder Constructors

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.

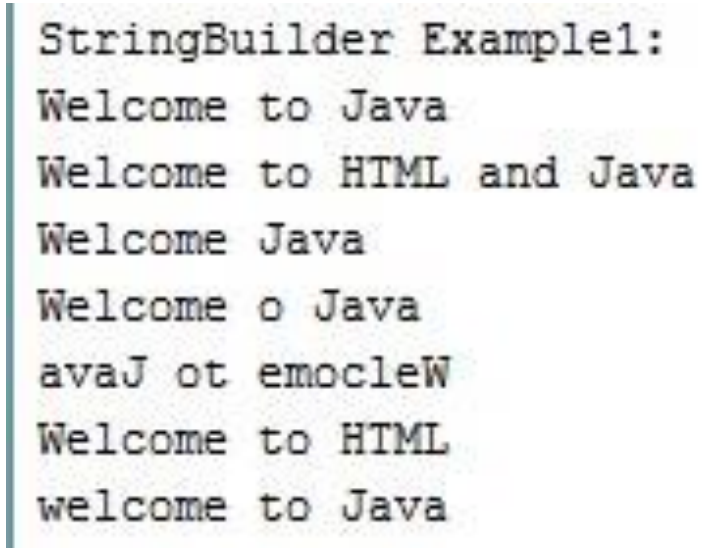
Modifying Strings in the Builder

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 (instance method)



```
public static void stringBuilder1(){
  System.out.println("StringBuilder Example1: ");
  StringBuilder stringBuilder = new StringBuilder("Welcome to ");
  stringBuilder.append("Java");
  System.out.println(stringBuilder.toString());
  //stringBuilder = new StringBuilder("Welcome to ");
  stringBuilder.insert(11, "HTML and ");
  System.out.println(stringBuilder.toString());
  stringBuilder = new StringBuilder("Welcome to Java");
  stringBuilder.delete(8, 11);
  System.out.println(stringBuilder.toString());
  stringBuilder = new StringBuilder("Welcome to Java");
  stringBuilder.deleteCharAt(8);
  System.out.println(stringBuilder.toString());
  stringBuilder = new StringBuilder("Welcome to Java");
  stringBuilder.reverse();
  System.out.println(stringBuilder.toString());
  stringBuilder = new StringBuilder("Welcome to Java");
  stringBuilder.replace(11, 15, "HTML");
  System.out.println(stringBuilder.toString());
  stringBuilder = new StringBuilder("Welcome to Java");
  stringBuilder.setCharAt(0, 'w');
  System.out.println(stringBuilder.toString());
}
```



```
StringBuilder Example1:
Welcome to Java
Welcome to HTML and Java
Welcome Java
Welcome o Java
avaJ ot emocleW
Welcome to HTML
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.

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.