AP Computer Science B

Java Object-Oriented Programming [Ver. 2.0]

Unit 4: Object-Oriented Design

WEEK 5: CHAPTER 11 OBJECT-ORIENTED THINKING (PART 2: LIBRARY)

DR. ERIC CHOU IEEE SENIOR MEMBER

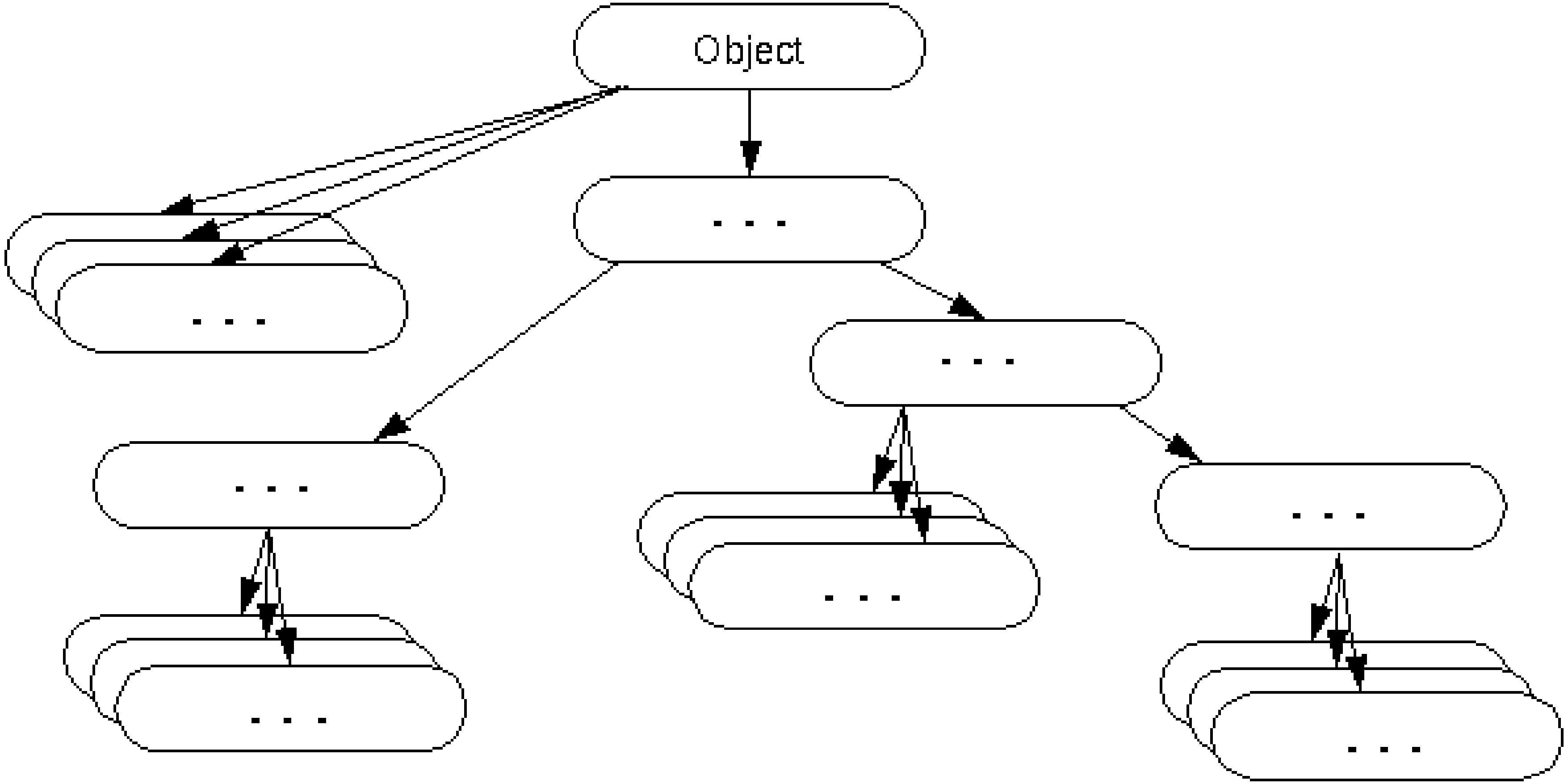
Objectives

- Object Class: Top of the object hierarchy. Default Inheritance (Generic Programming, Generalization)
- Overloading and Generalization
- Use of this
- Use of Library:
  - Numerical Computation
  - Text Processing

Object Class

LECTURE 1

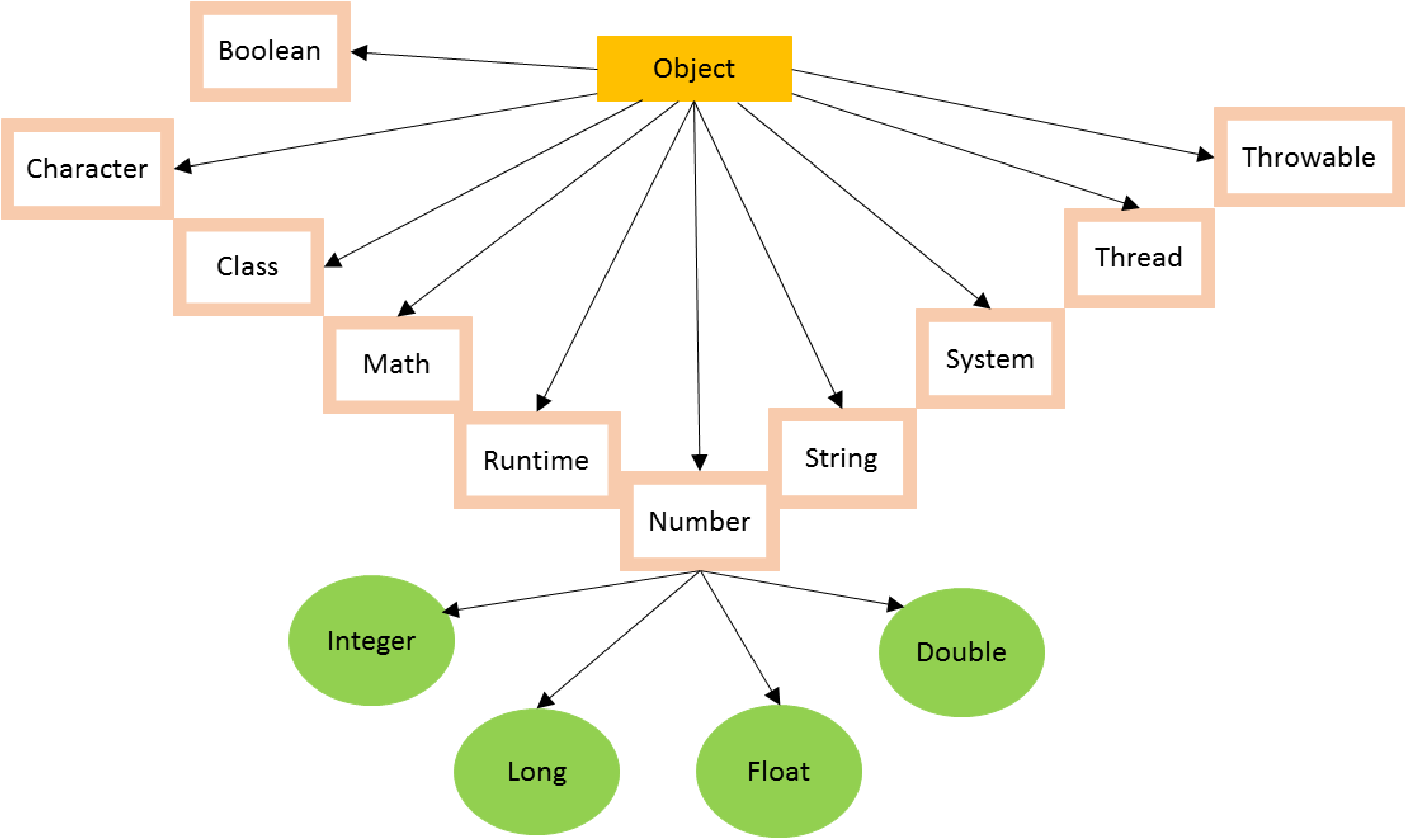
Every Class Inherits from Object Class



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()**

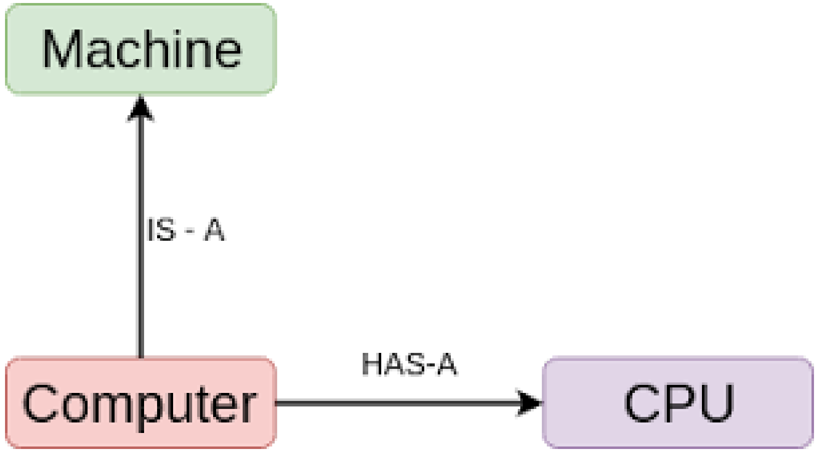
  Returns a string representation of the object.

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 2

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

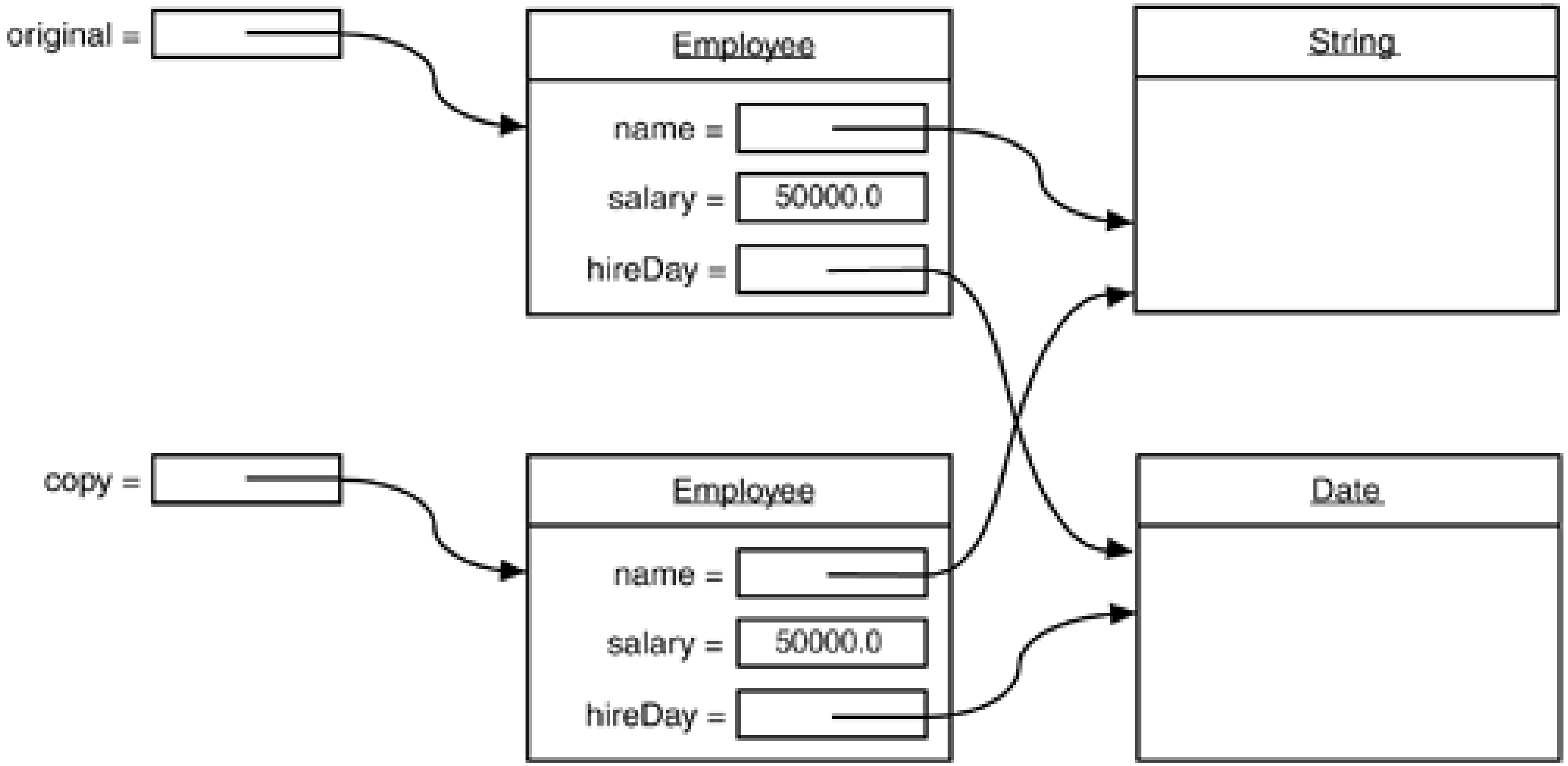
  protected Object clone() throws CloneNotSupportedException

  or:

  public Object clone() throws CloneNotSupportedException

clone() method

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

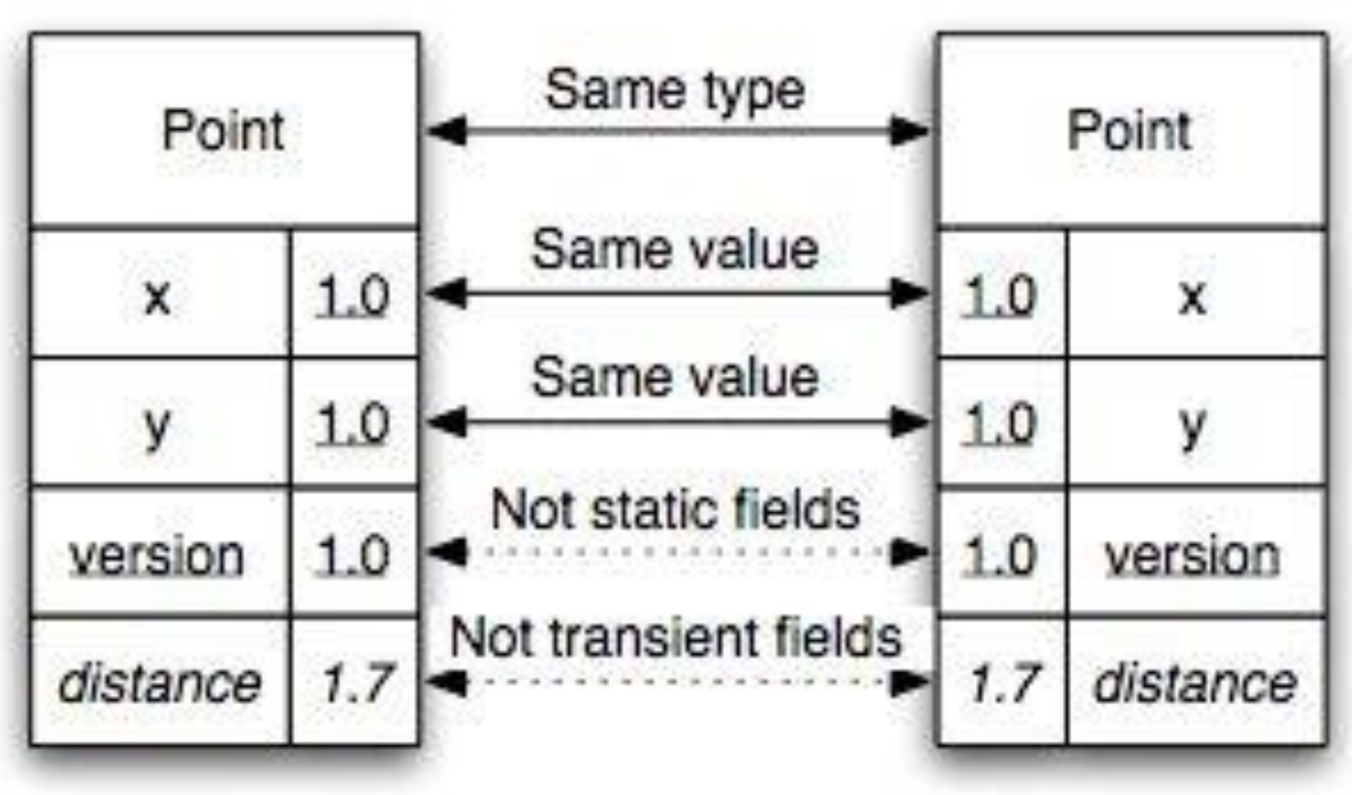


The equals() Method

- 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()**:

```
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 **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.

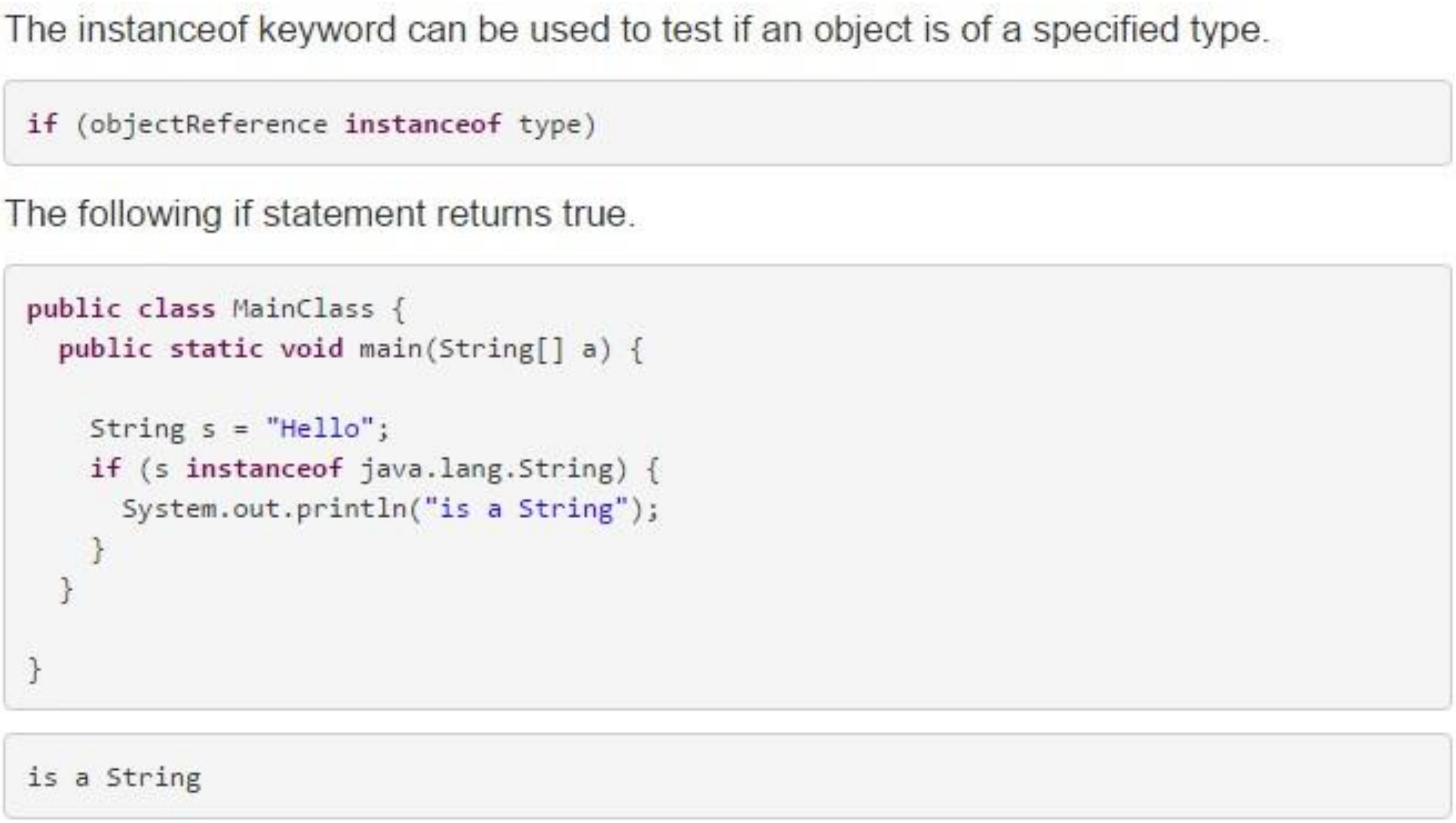
object instanceof class will return a boolean value whether the **object** is of the **class**.

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

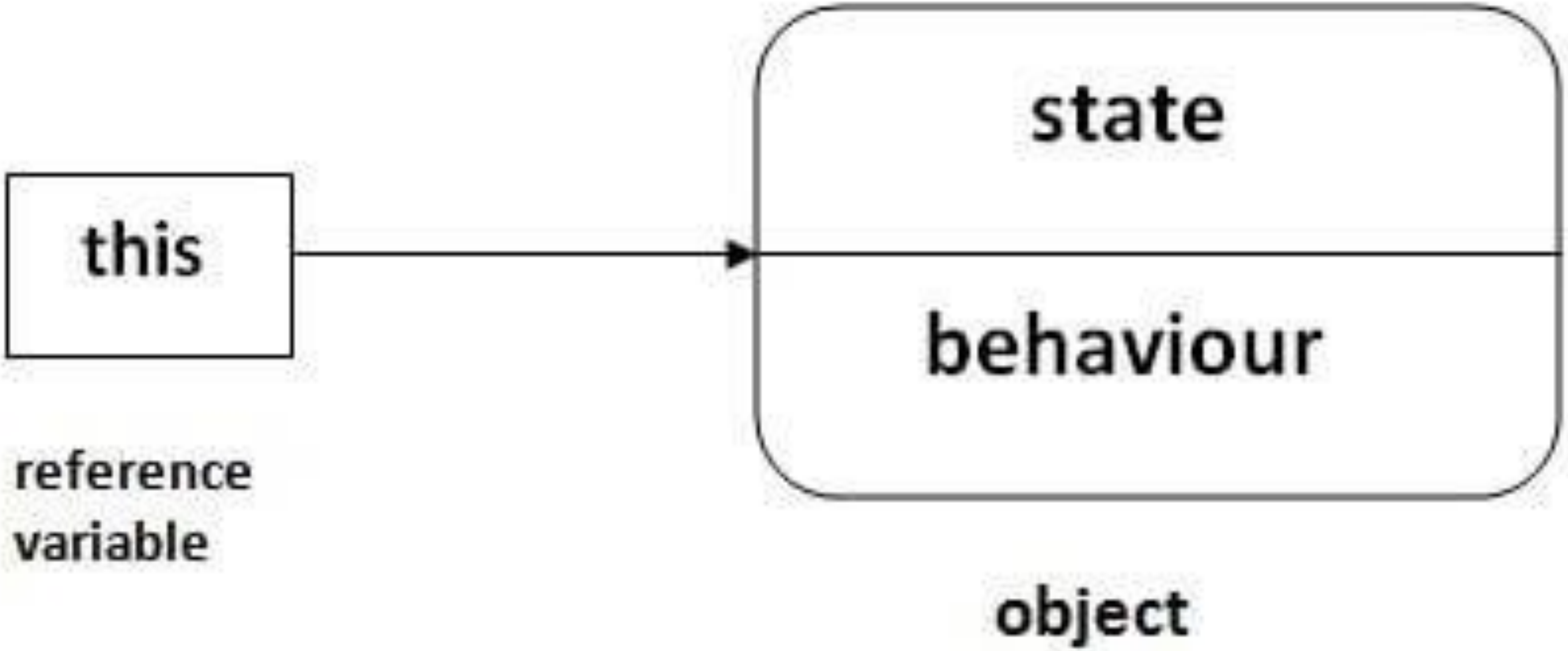
- The Object class provides a callback method, **finalize()**, that may be invoked on an object when it becomes garbage. Object's implementation of **finalize()** does nothing—you can override **finalize()** to do cleanup, such as freeing resources.
- The **finalize()** method may be called automatically 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.

Use of this Reference

LECTURE 3

Use of this reference

- The pointer to the object in the heap memory for the current object.
- Reference to current object: this.x;
- Reference to constructor this(1.0);
- Calling a method: this.getArea();



this

reference
variable

state

behaviour

object

Calling overloading constructor

shorter constructor calling longer constructor method

(1) eliminate the need to re-write different format of constructors.

(2) Write the longer constructor first. Then, write constructors of all possible lengths.

(3) Higher maintainability

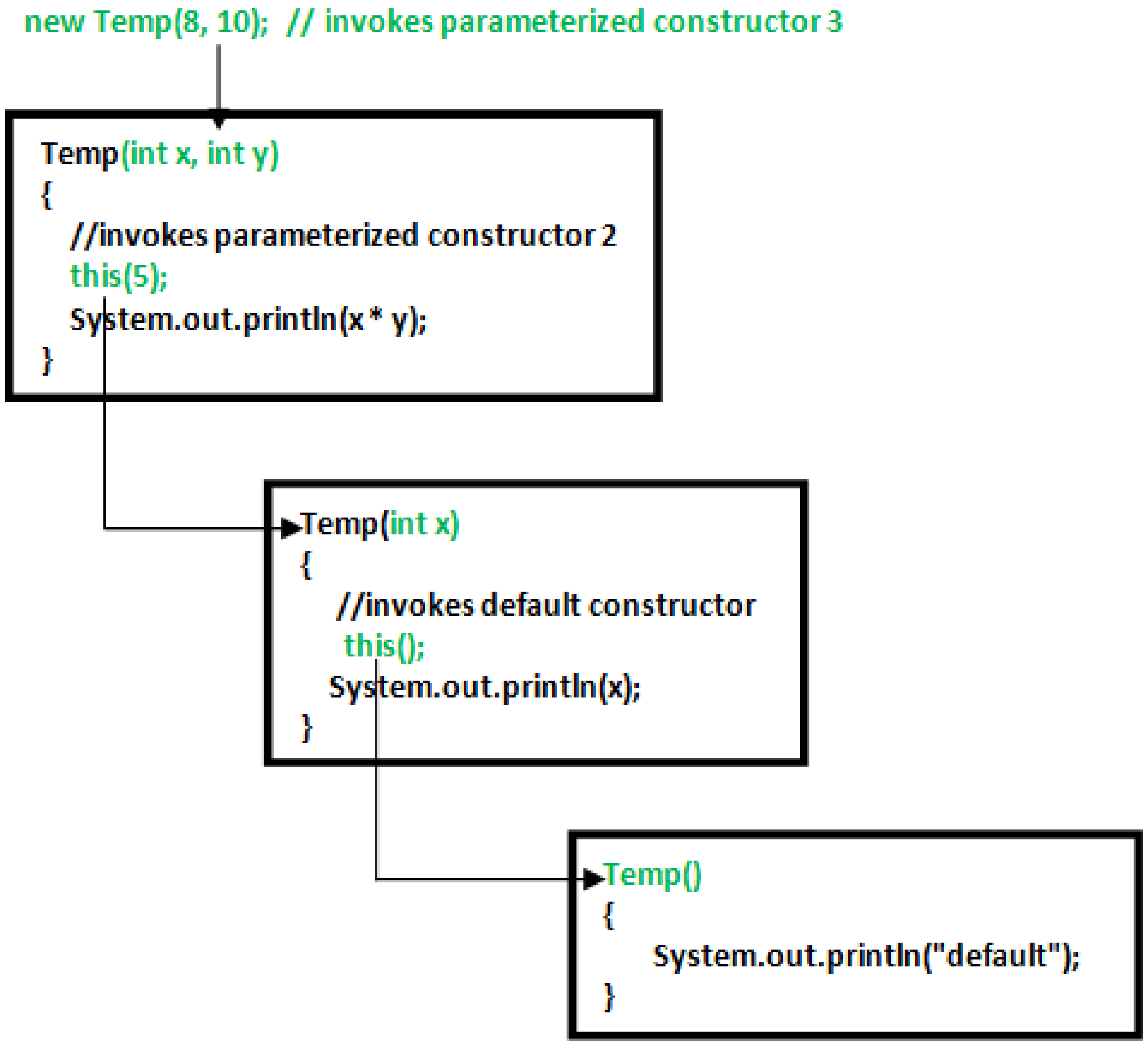
Calling Overloaded Constructor

```
public class Circle {
  private double radius;

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

  public Circle() {
    this(1.0);
  }

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

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

this is used to invoke another constructor

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

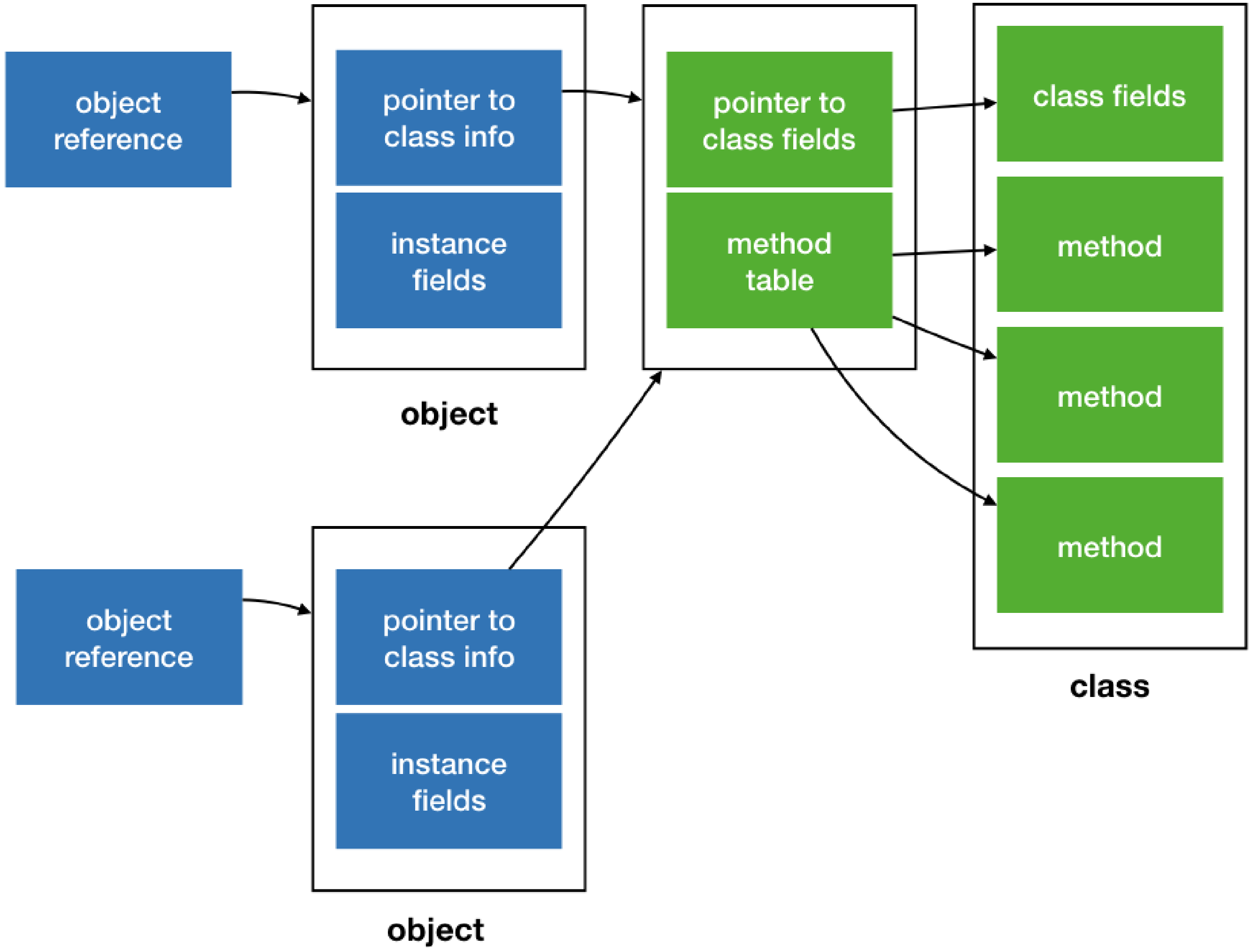


```
new Temp(8, 10);  // invokes parameterized constructor 3
```

```
Temp(int x, int y)
{
   //invokes parameterized constructor 2
   this(5);
   System.out.println(x * y);
}
```

```
Temp(int x)
{
    //invokes default constructor
     this();
   System.out.println(x);
}
```

```
Temp()
{
      System.out.println("default");
}
```

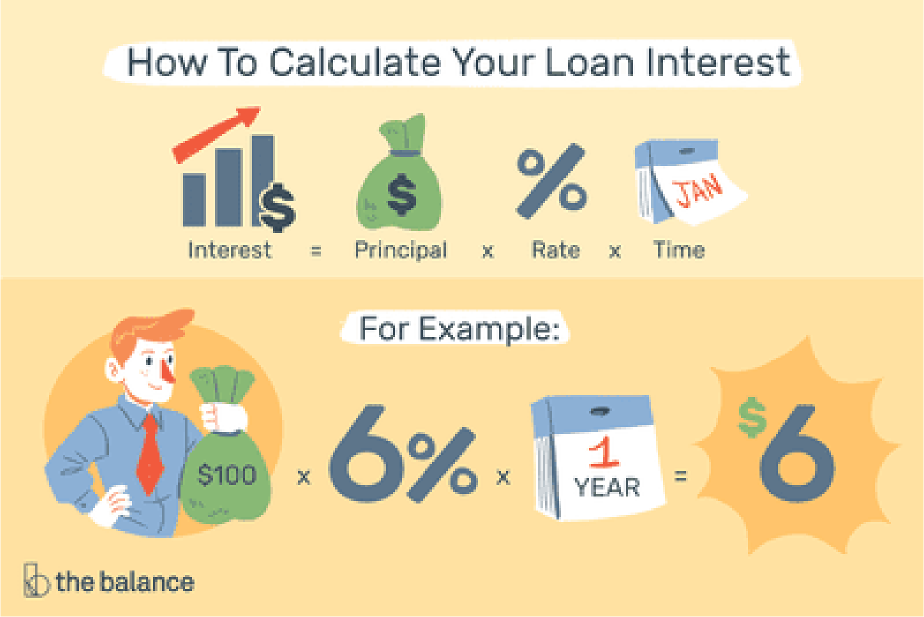

object reference
pointer to class info
instance fields
object
pointer to class fields
method table
class fields
method
method
method
class
object reference
pointer to class info
instance fields
object

Demo Program: Loan Class

LECTURE 4

Objective

- Comparison of a Structured program and a OOP program
- Demonstration of usage for Math API


How To Calculate Your Loan Interest
Interest = Principal x Rate x Time
JAN
For Example:
$100 x 6% x 1 YEAR = $6
the balance

Math API

- Besides text processing using String class, numerical computation is another application field that commonly encountered by programmers.
- This lecture, we use loan payment calculation as an example to demonstrate the techniques for developing mathematical computation both in **structural programming** and in **object-oriented programming**.

Structural Programming

COMPUTELOAN.JAVA

System Analysis

The output is the monthly payment and total payment, which can be obtained using the following formulas:

$$monthlyPayment = \frac{loanAmount * monthlyInterestRate}{1 - \frac{1}{(1 + monthlyInterestRate)^{(numberOfYears * 12)}}}$$

So, the input needed for the program is the monthly interest rate, the length of the loan in years, and the loan amount.

Loan Payment Calculation Formula

(For the derivation of the formula: check MortgageLoanDerivation.pdf)

$$P = \frac{r(PV)}{1-(1+r)^{-n}}$$

$$P = Payment$$
$$PV = Present\ Value$$
$$r = rate\ per\ period$$
$$n = number\ of\ periods$$

Mathematical Model

```
double monthlyPayment;          // P  (output)
double loanAmount;              // PV (input)
double monthlyInterestRate;     // r   (input)
double numberOfYears * 12;      // n (number of months: input)

monthlyPayment = loanAmount * monthlyInterestRate /
(1 - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
```

Java Model

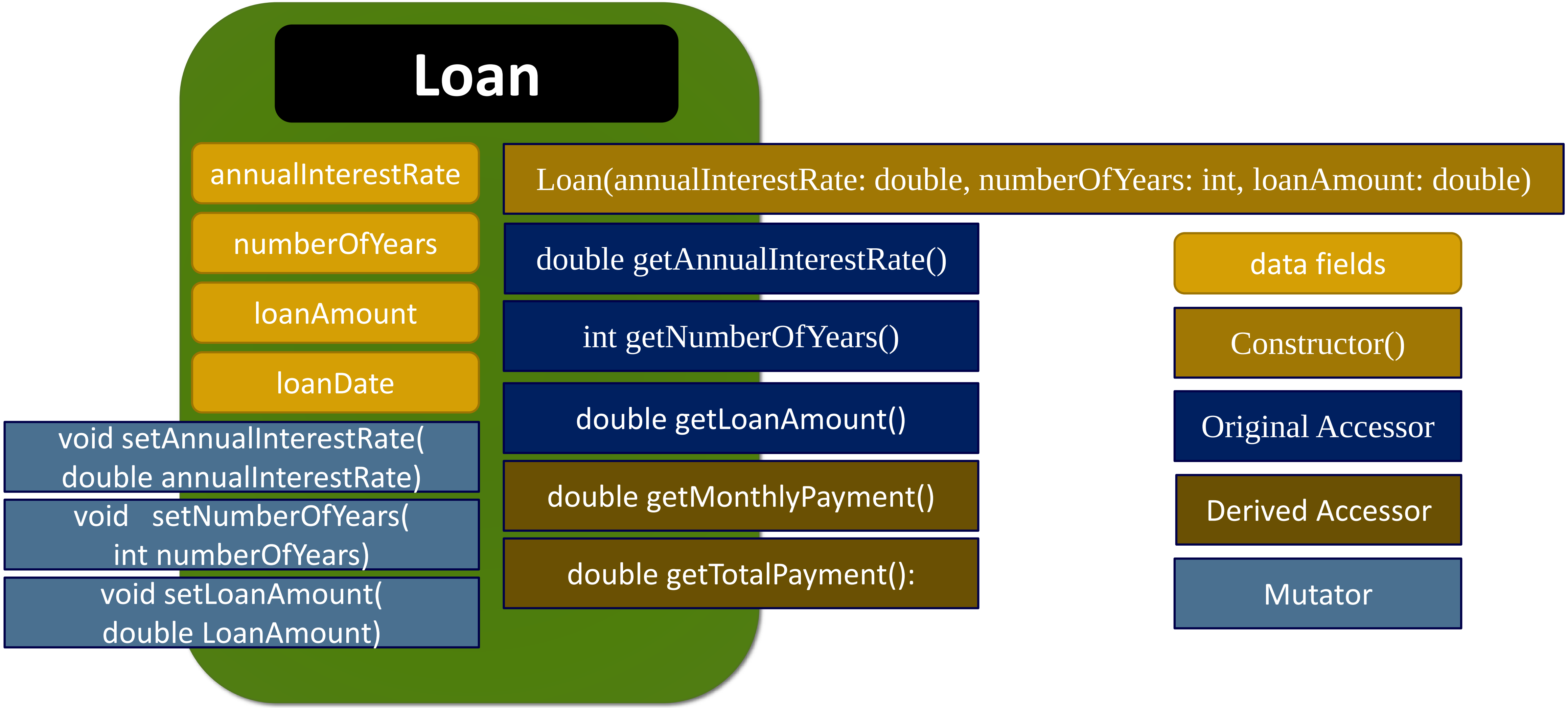
Demonstration Program

COMPUTELOAD.JAVA

Object-Oriented Programming

LOAN.JAVA AND TESTLOAN.JAVA

Loan Class



Loan

annualInterestRate

numberOfYears

loanAmount

loanDate

void setAnnualInterestRate(
double annualInterestRate)

void setNumberOfYears(
int numberOfYears)

void setLoanAmount(
double LoanAmount)

Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)

double getAnnualInterestRate()

int getNumberOfYears()

double getLoanAmount()

double getMonthlyPayment()

double getTotalPayment():

data fields

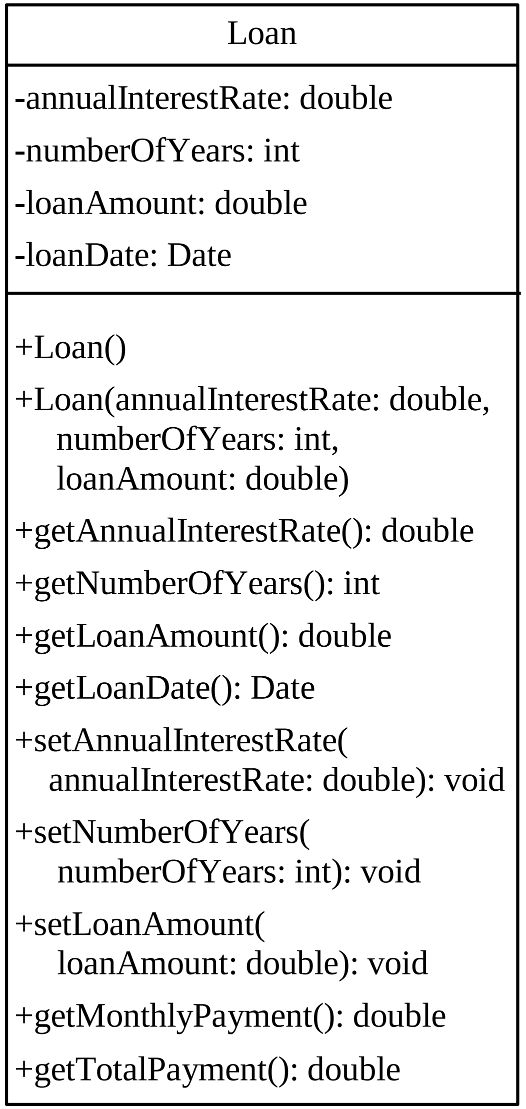
Constructor()

Original Accessor

Derived Accessor

Mutator

Designing the Loan Class



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

Demonstration Program

LOAN.JAVA TESTLOAN.JAVA

Numerical vs. Analytical Methods

Analytical Methods

- Solution have been derived for some engineering problems using analytical (or exact) methods.
- In general there are few closed-form engineering or exact solutions including problems that can be approximated with linear models or that have simple geometry and low dimensionality.
- These solutions are often useful and provide excellent insight into the behavior of an engineering system.

Analytical vs. Numerical methods

Need for Numerical Methods

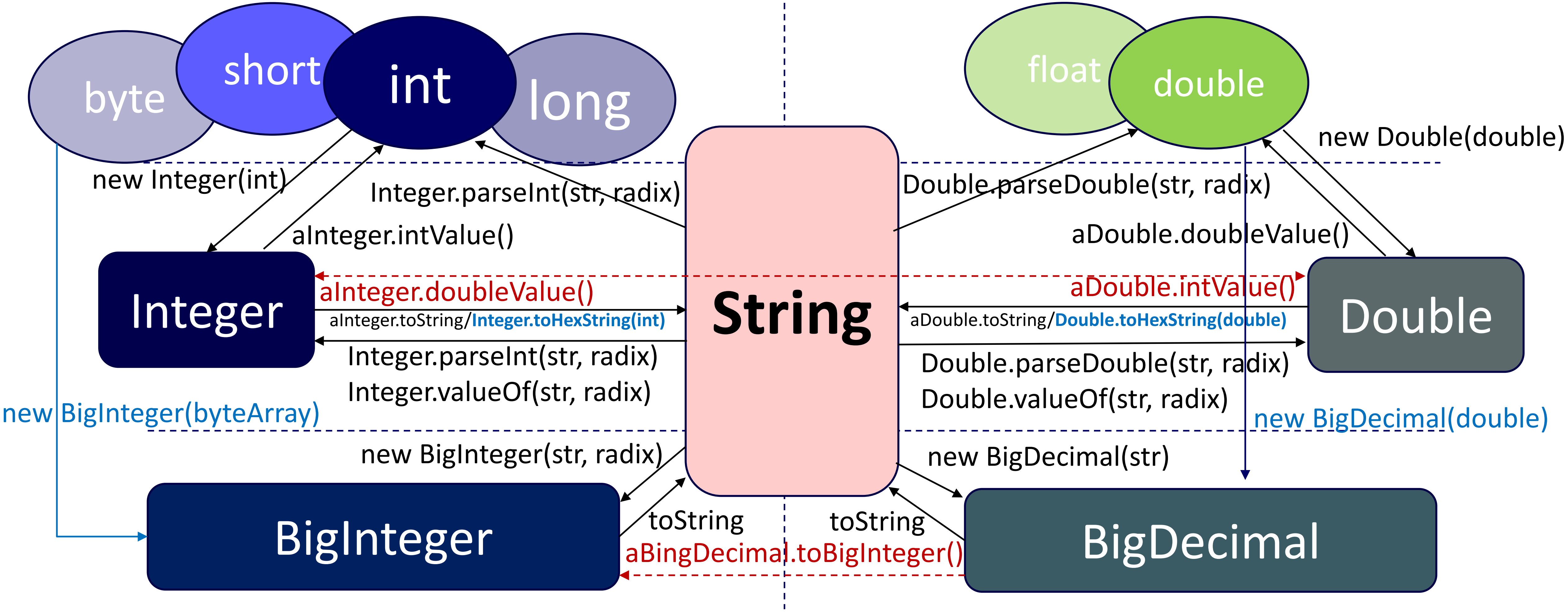
- In general, there are few analytical (closed-form) solutions for many practical engineering problems.
- Numerical methods can handle:
  - Large systems of equations
  - Non-linearity
  - Complicated geometries that are common in engineering practice and that are often impossible to solve analytically.

Examples: $F=\int_0^{30}\left(\frac{\cos(z)+z}{5+z}\right)e^{-2z/30}dz$ $\frac{x}{1+\sin(x)}+e^x=0$

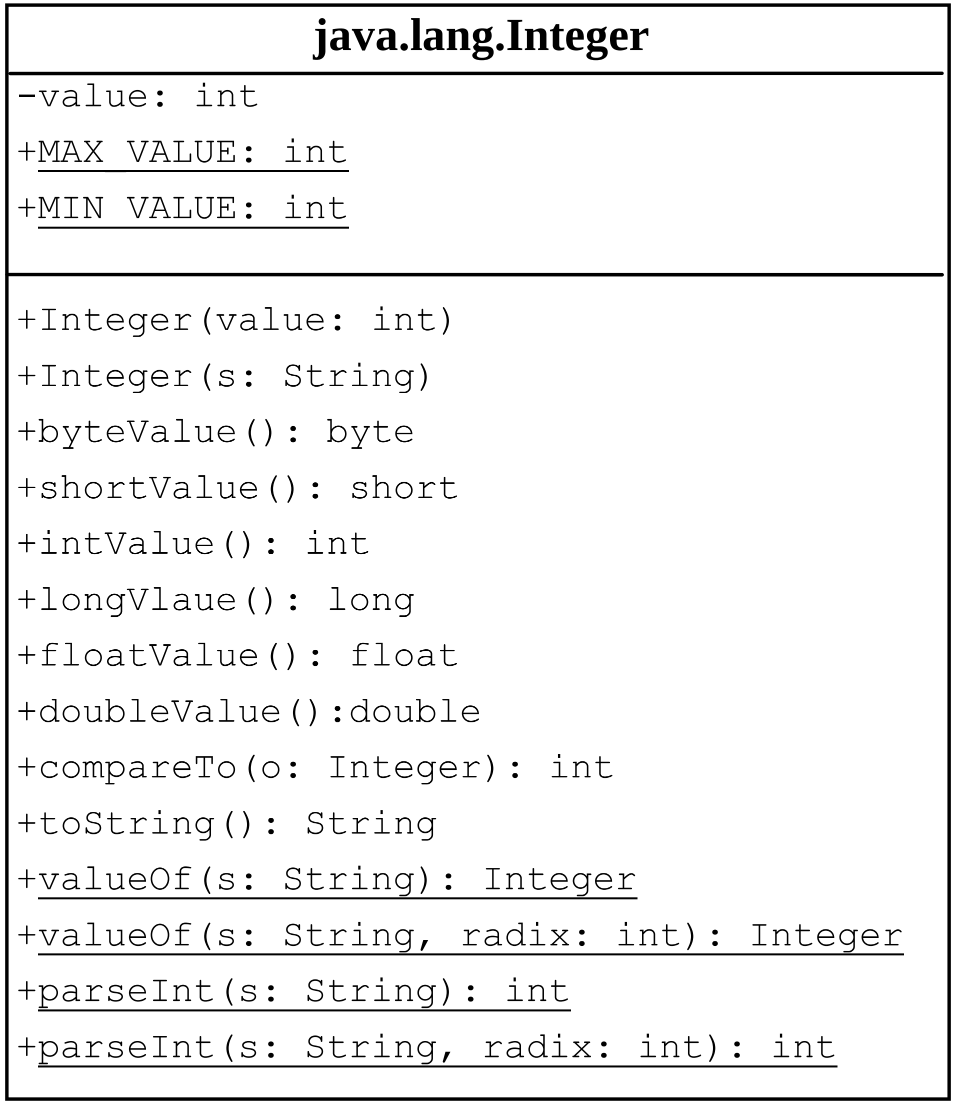
Math Processing I: Data/Object Type Conversion

LECTURE 5

Map for Java Number Space

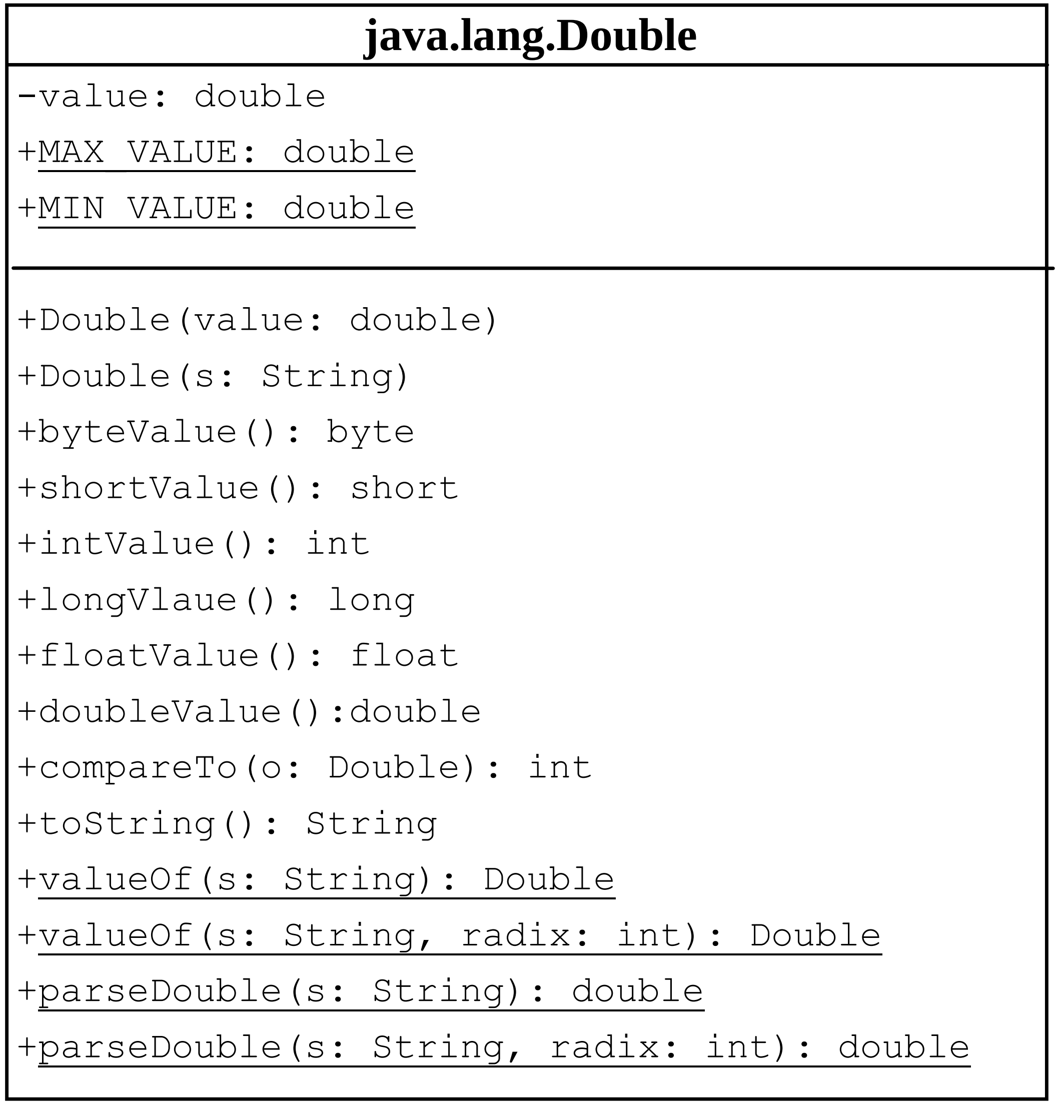

byte
short
int
long
float
double
new Integer(int)
Integer.parseInt(str, radix)
aInteger.intValue()
Double.parseDouble(str, radix)
aDouble.doubleValue()
new Double(double)
Integer
String
Double
aInteger.doubleValue()
aInteger.toString/Integer.toHexString(int)
Integer.parseInt(str, radix)
Integer.valueOf(str,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

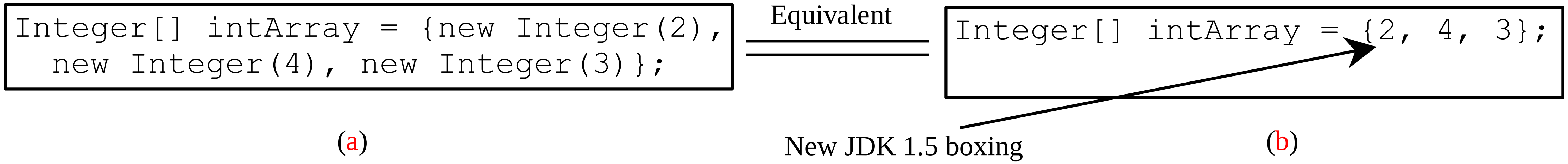
```
Double doubleObject = Double.valueOf("12.4");

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)};
```

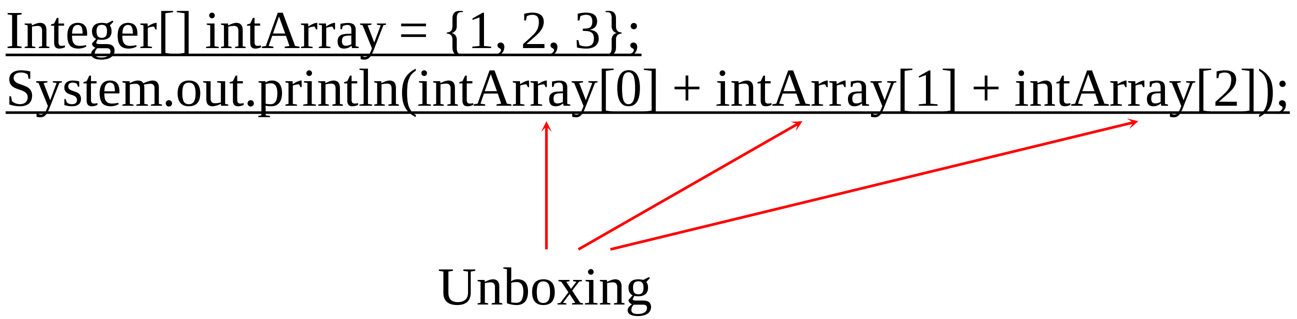
Equivalent

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

(a)

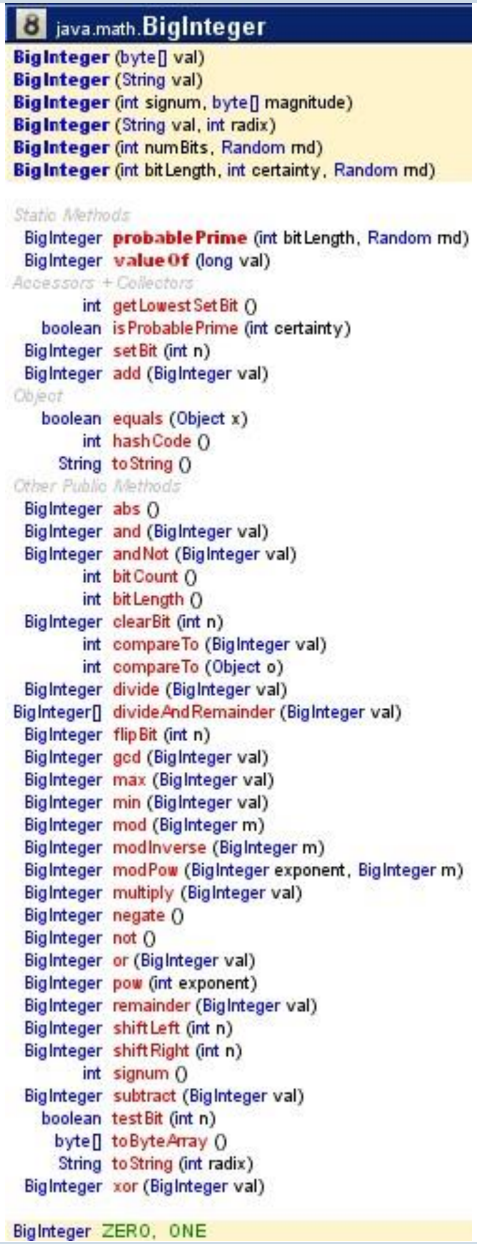
New JDK 1.5 boxing

(b)

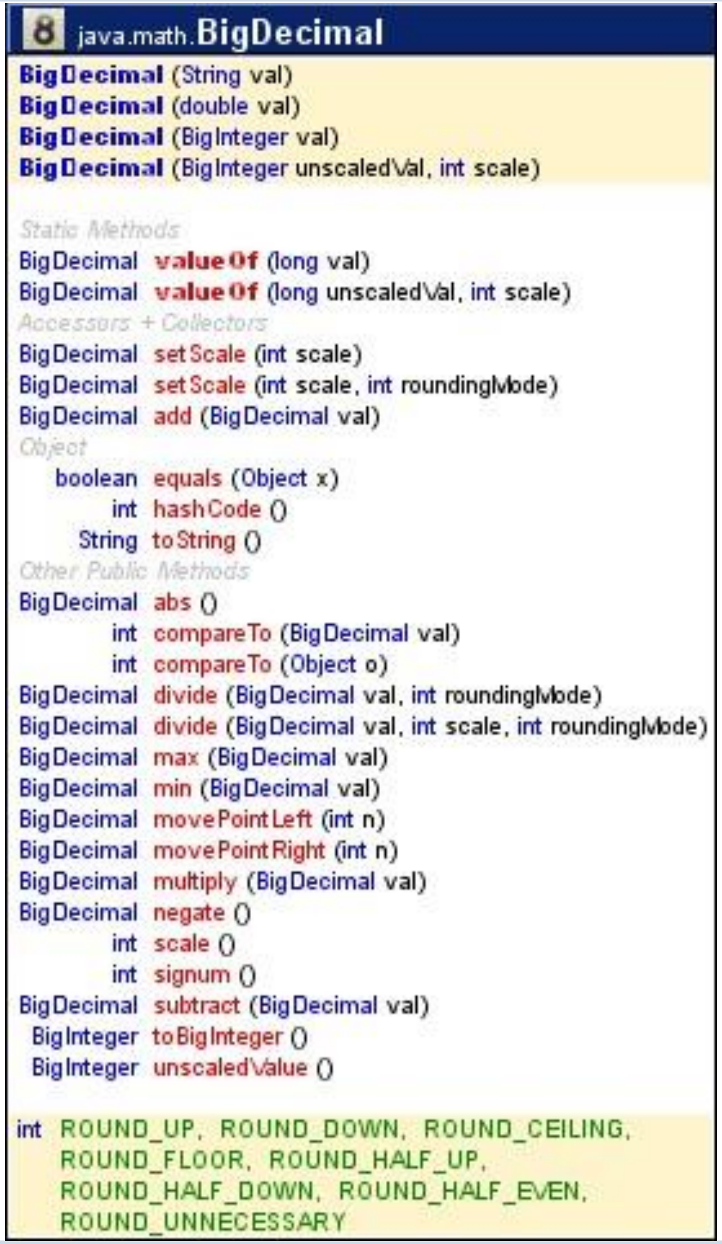


```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

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);
```

String Processing I: Extra String Programs

LECTURE 6

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.

```
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]

Lab Project:

Finish this project within 25 minutes (Requirement for AP EXAM is 22.5 min per FRQ problem).

Please pause here before proceeding to this problem solution part.

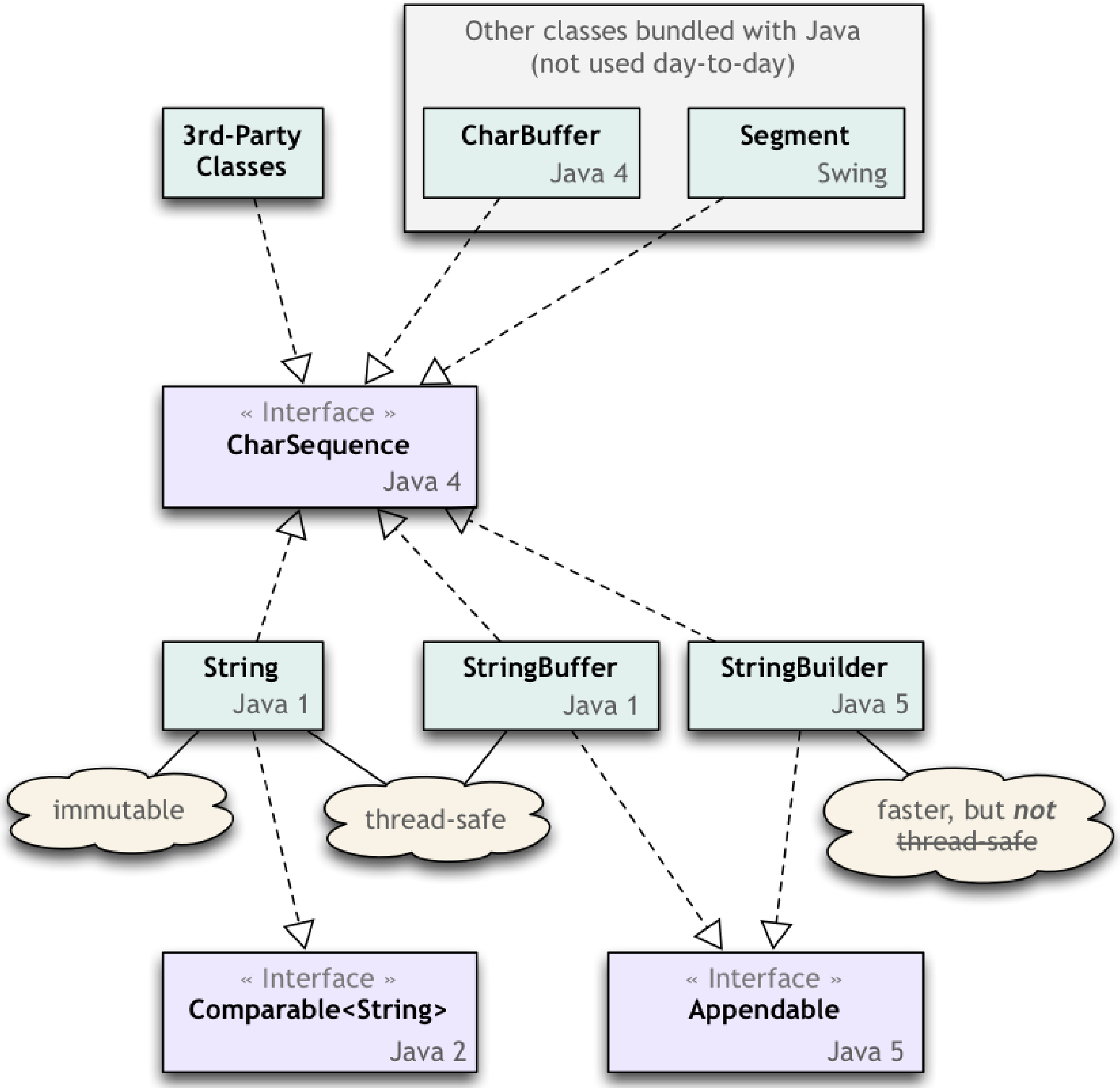
Project: StringList.java

Student should work on this project in Class.

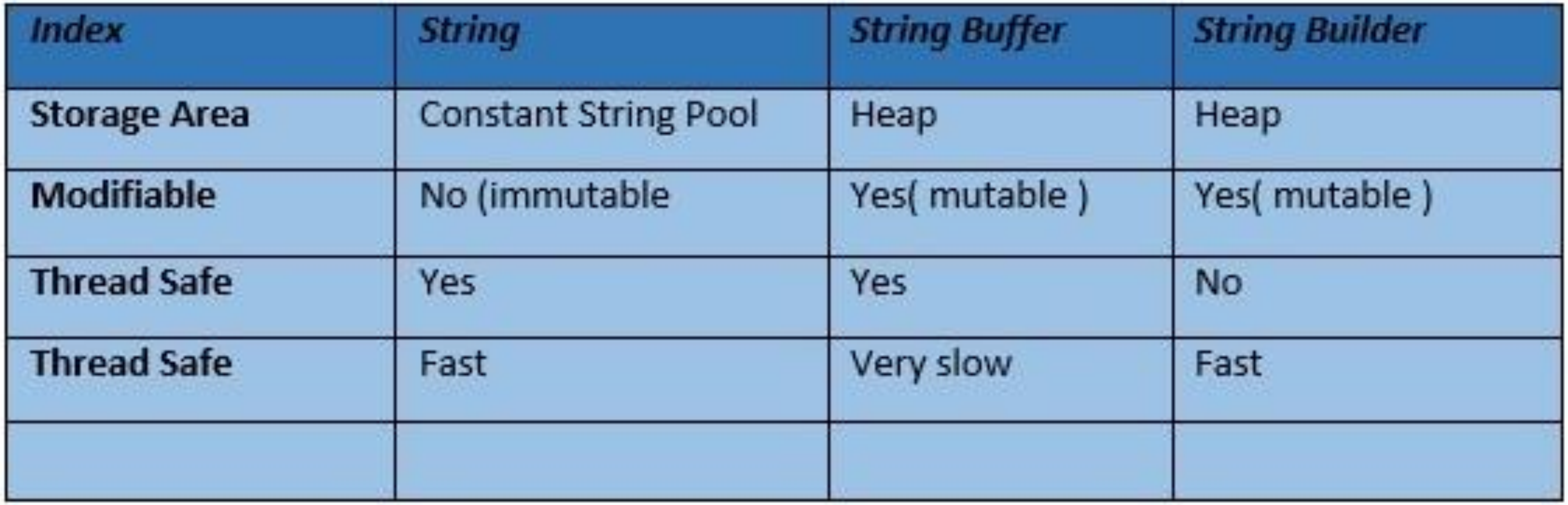
String, StringBuilder, and StringBuffer

LECTURE 7

String Types In Java 8

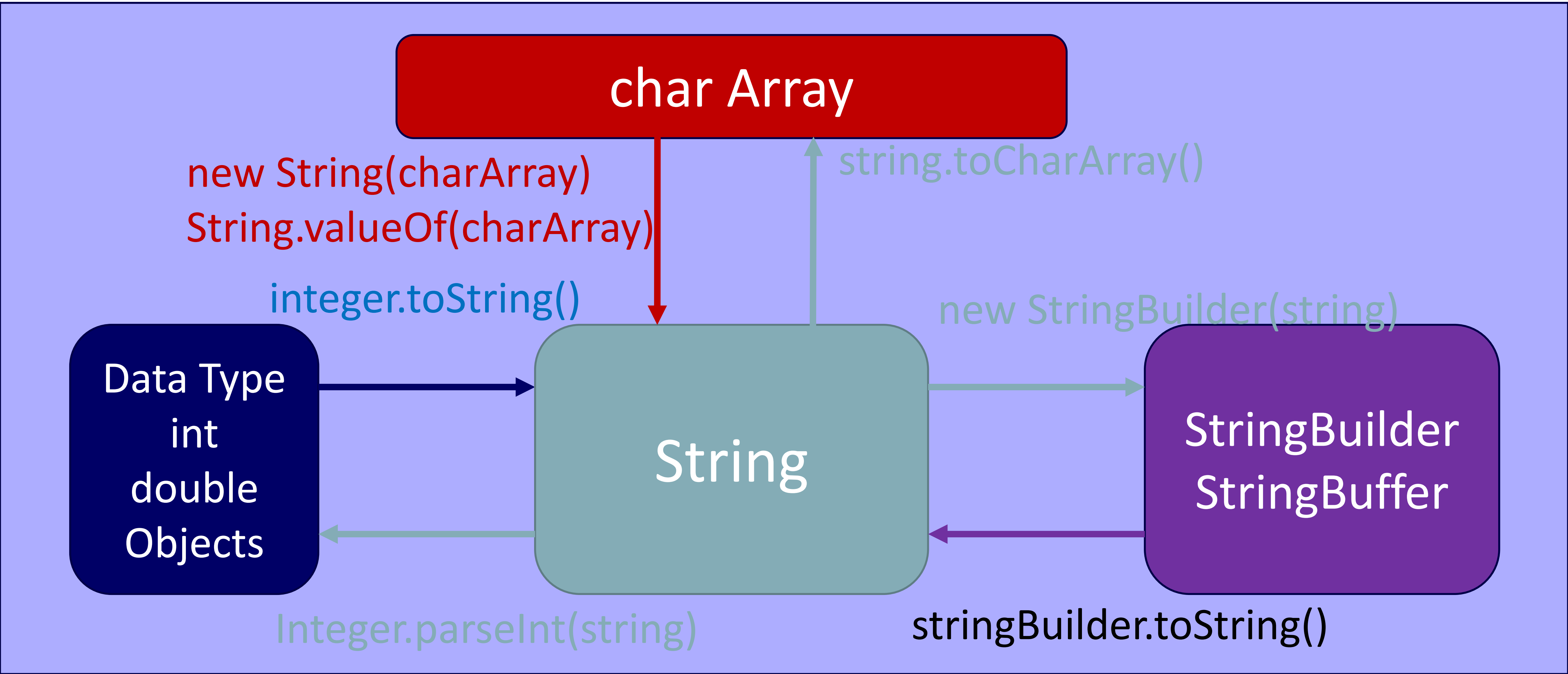


String <-> StringBuilder (StringBuffer)



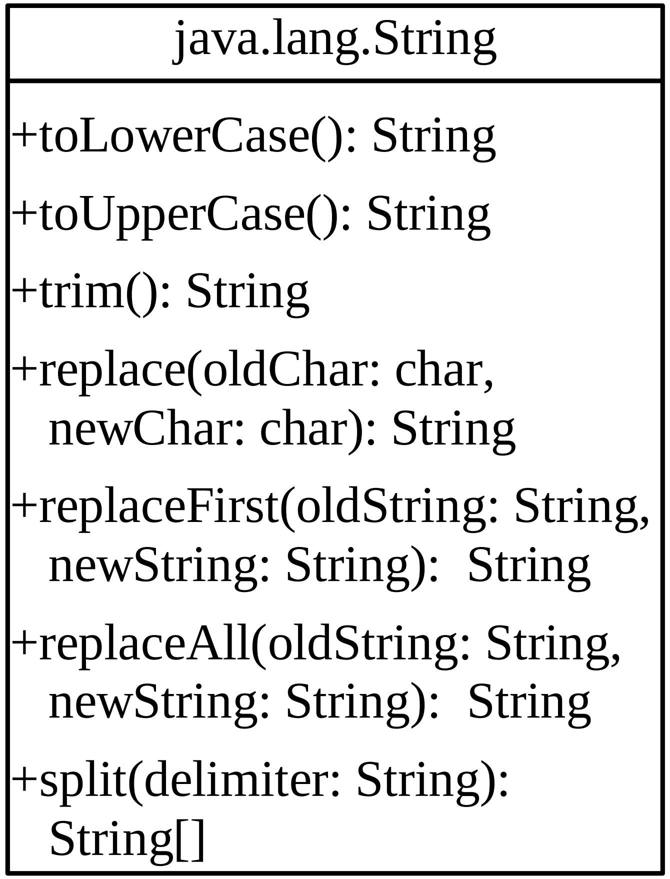
Index	*String*	*String Buffer*	*String Builder*
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable	Yes(mutable)	Yes(mutable)
Thread Safe	Yes	Yes	No
Thread Safe	Fast	Very slow	Fast

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 ♠ ♠
```

Note that

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

is equivalent to

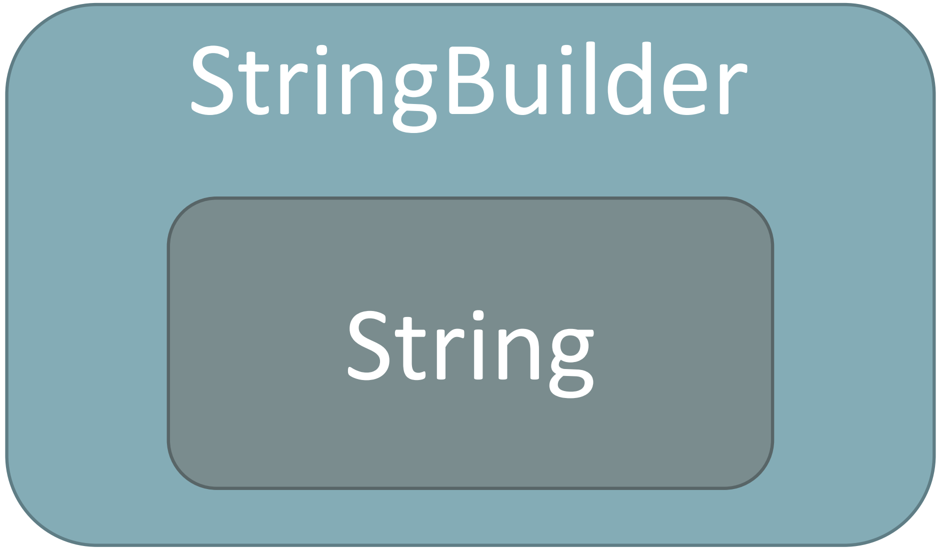
```
System.out.print(String.format(format, item1, item2, ..., itemk));
```

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

String, StringBuilder, and StringBuffer II

LECTURE 8

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)

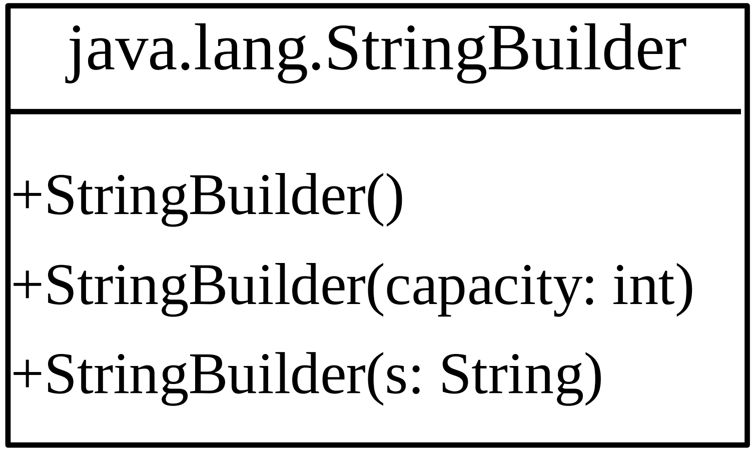
charAt(index): like get in ArrayList

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.

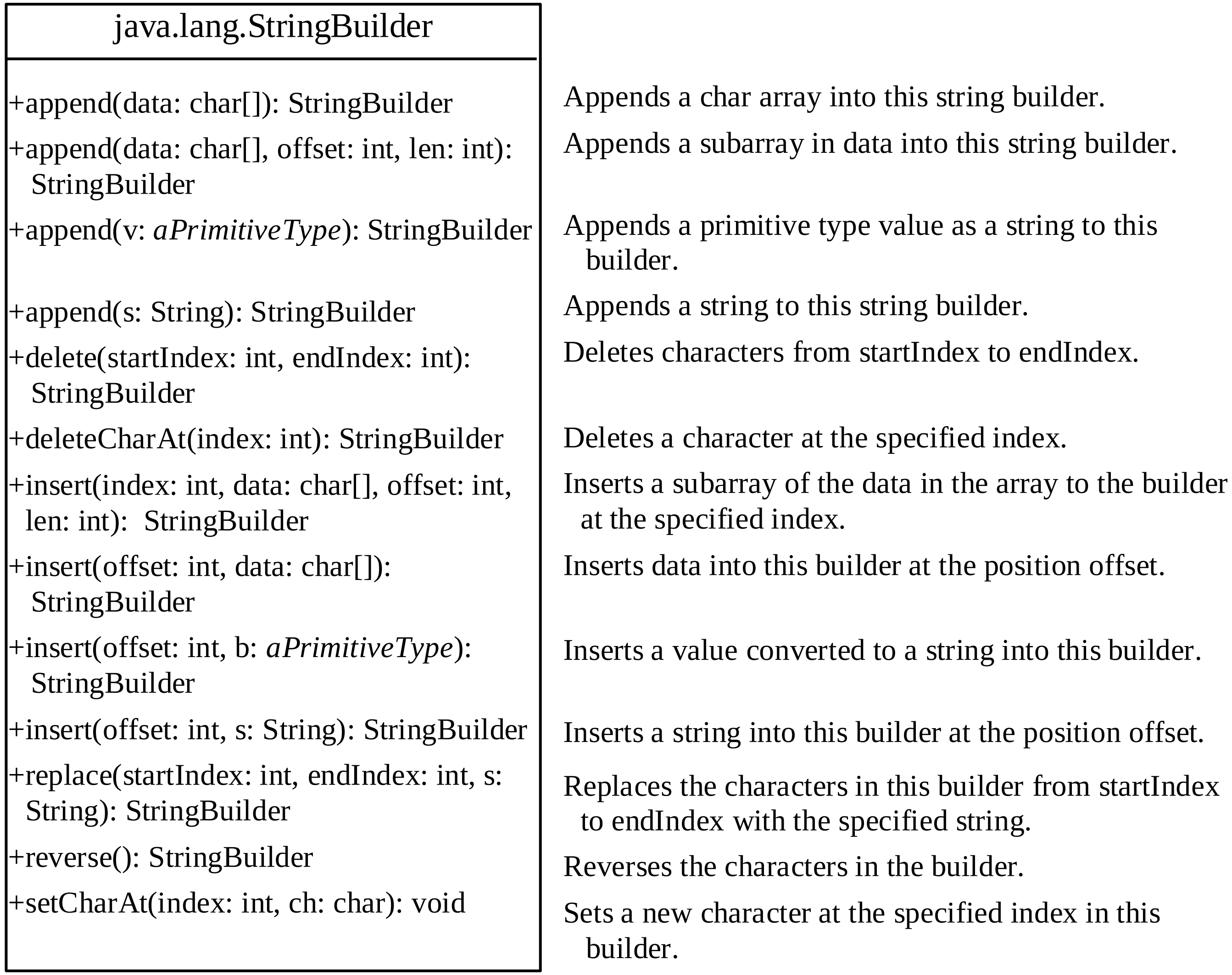
StringBuilder Constructors



java.lang.StringBuilder

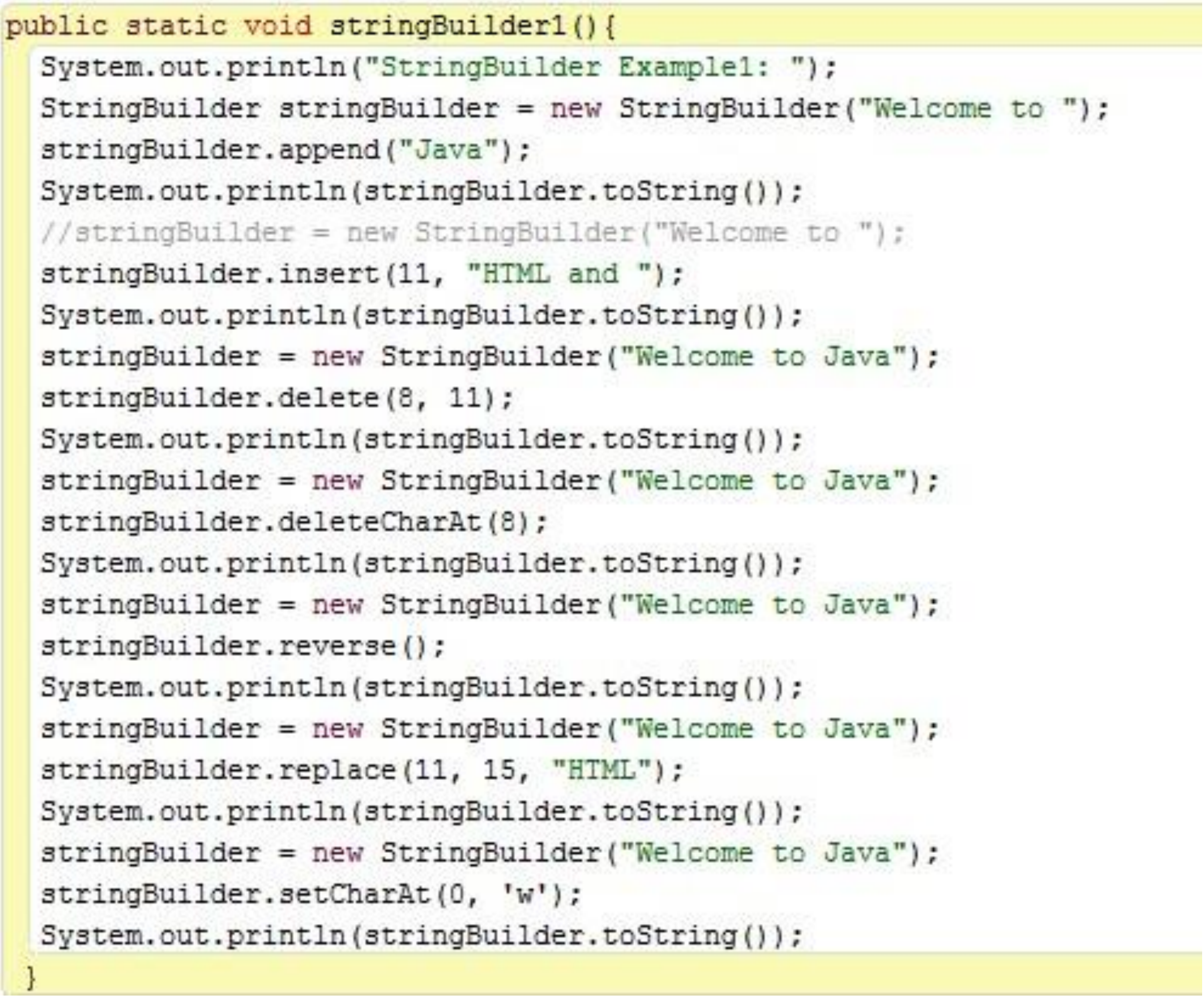
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

Modifying Strings in the Builder

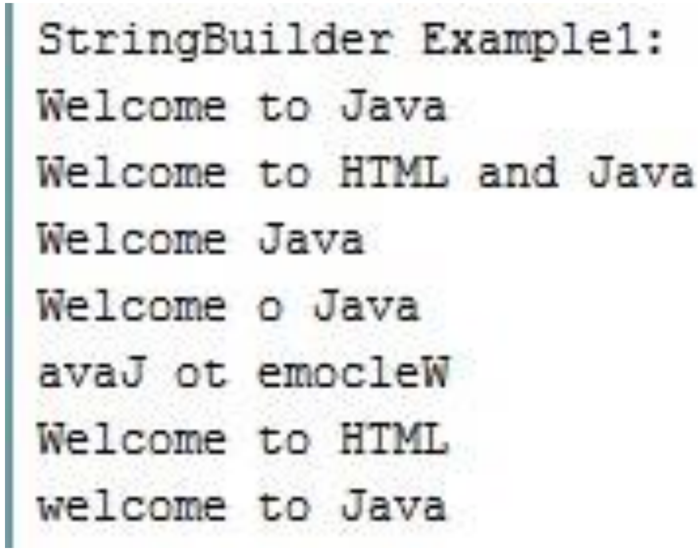


java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a 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)

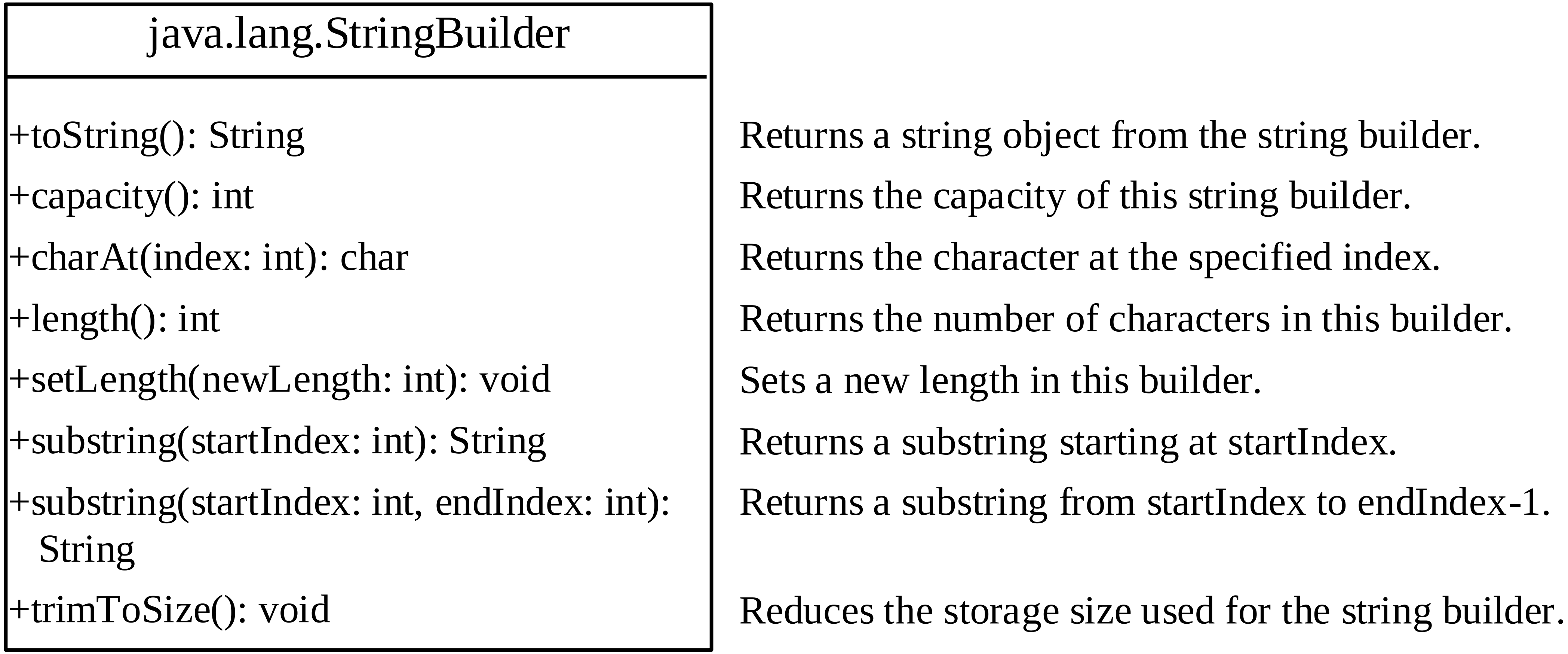


```
public static void stringBuilder1(){
    System.out.println("StringBuilder Example1: ");
    StringBuilder stringBuilder = new StringBuilder("Welcome to ");
    stringBuilder.append("Java");
    System.out.println(stringBuilder.toString());
    //stringBuilder = new StringBuilder("Welcome to ");
    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.

Demo Program: Check Palindrome

LECTURE 9

Problem: Finding Palindromes

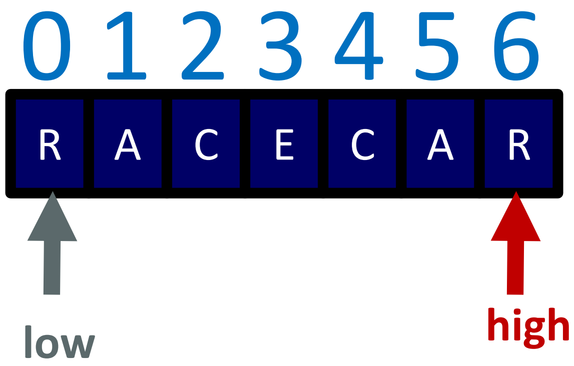
Palindrome.java

- Objective: Checking whether a string is a palindrome: a string that reads the same forward and backward.

STRESSED is Desserts SPELLED BACKWARDS
"STRESSED" is "DESSERTS" spelled backwards
STRESSED SPELLED BACKWARD IS DESSERTS
STRESSED is JUST DESSERTS SPELLED BACKWARDS
stressed
desserts
STRESSED spelled backward is DESSERTS

Bi-directional Traversal

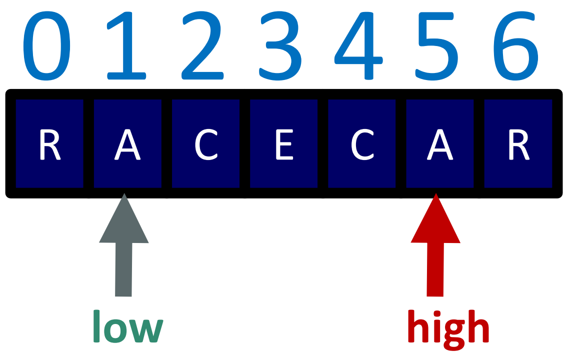
```
String s = input.nextLine();
int low = 0;
int high = s.length() - 1;
boolean isPalindrome = true;
while (low < high) {
  if (s.charAt(low) != s.charAt(high)) {
    isPalindrome = false;
    break;
  }
  low++;
  high--;
}
```



0	1	2	3	4	5	6
R	A	C	E	C	A	R

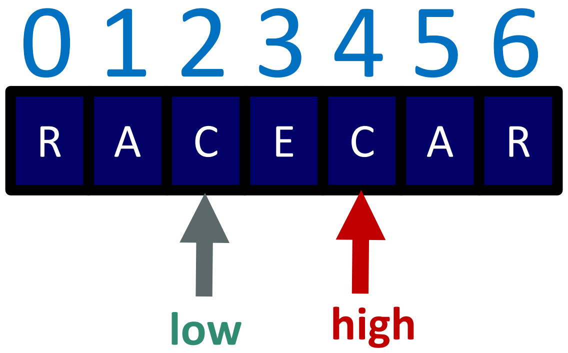
Bi-directional Traversal

```
String s = input.nextLine();
int low = 0;
int high = s.length() - 1;
boolean isPalindrome = true;
while (low < high) {
  if (s.charAt(low) != s.charAt(high)) {
    isPalindrome = false;
    break;
  }
  low++;
  high--;
}
```



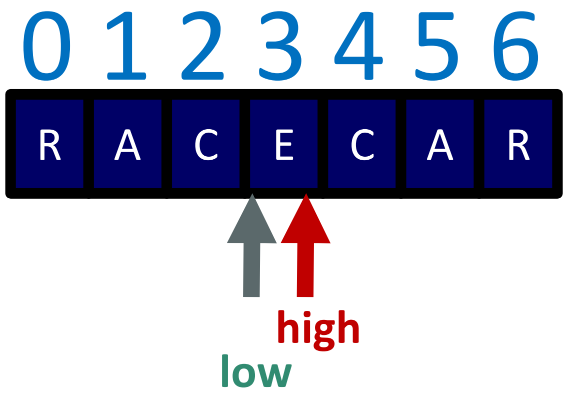
Bi-directional Traversal

```
String s = input.nextLine();
int low = 0;
int high = s.length() - 1;
boolean isPalindrome = true;
while (low < high) {
  if (s.charAt(low) != s.charAt(high)) {
    isPalindrome = false;
    break;
  }
  low++;
  high--;
}
```



Bi-directional Traversal

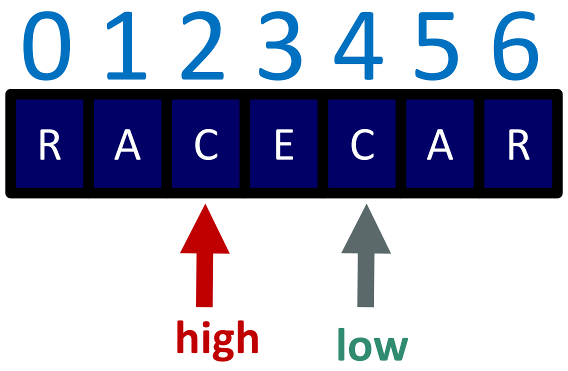
```
String s = input.nextLine();
int low = 0;
int high = s.length() - 1;
boolean isPalindrome = true;
while (low < high) {
  if (s.charAt(low) != s.charAt(high)) {
    isPalindrome = false;
    break;
  }
  low++;
  high--;
}
```



0	1	2	3	4	5	6
R	A	C	E	C	A	R

Bi-directional Traversal

```
String s = input.nextLine();
int low = 0;
int high = s.length() - 1;
boolean isPalindrome = true;
while (low < high) {
  if (s.charAt(low) != s.charAt(high)) {
    isPalindrome = false;
    break;
  }
  low++;
  high--;
}
```

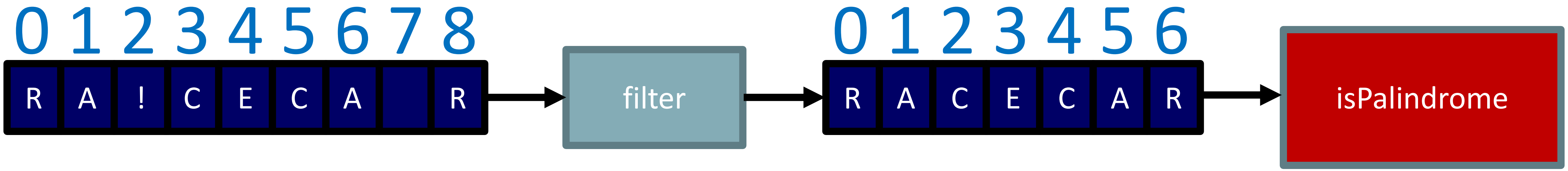


0	1	2	3	4	5	6
R	A	C	E	C	A	R

Stop

Problem: Checking Palindromes Ignoring Non-alphanumeric Characters

This example gives a program that counts the number of occurrence of each letter in a string. Assume the letters are not case-sensitive.



0	1	2	3	4	5	6	7	8
R	A	!	C	E	C	A		R

0	1	2	3	4	5	6
R	A	C	E	C	A	R

String, StringBuilder API Methods Used

Character Wrapper Class:

isLetterOrDigit()

String Class:

equals()

StringBuilder Class (Wrapper Class of String): (Non-AP Topic)

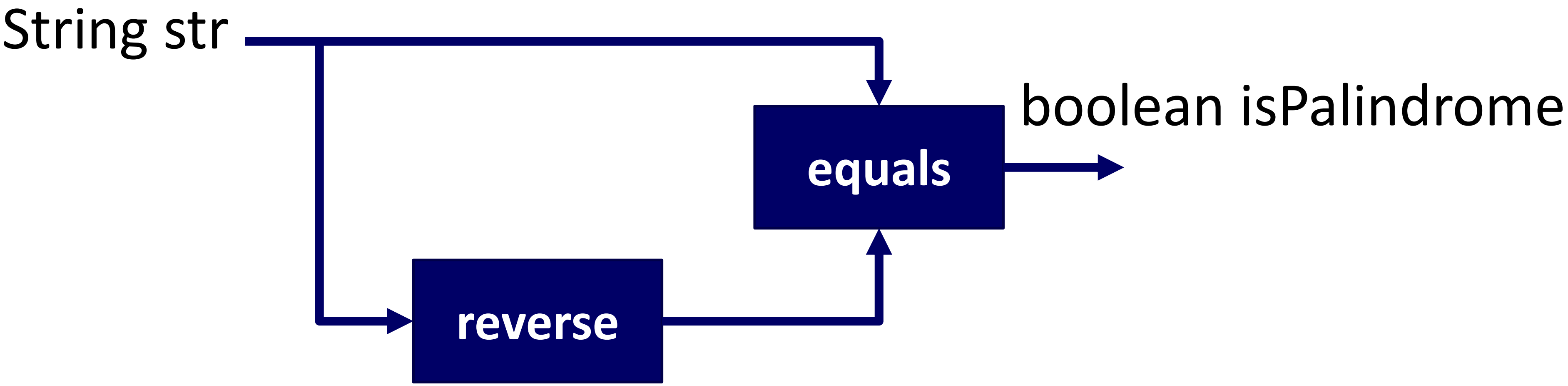
reverse()

toString()

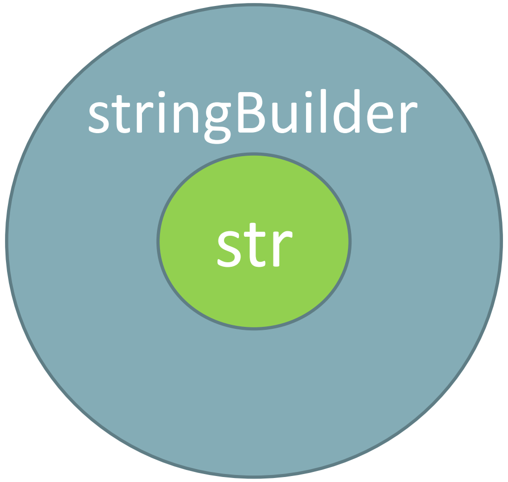
append(): similar to add in ArrayList Class

Algorithm

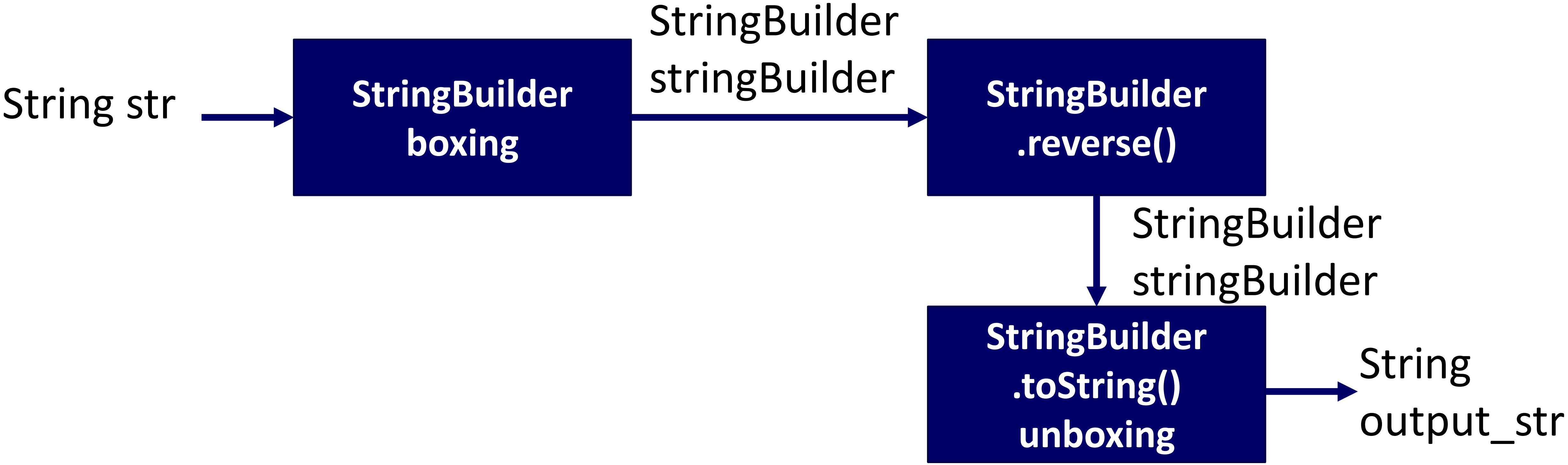
isPalindrome():



Algorithm



reverse():



String str

StringBuilder boxing

StringBuilder stringBuilder

StringBuilder .reverse()

StringBuilder stringBuilder

StringBuilder .toString() unboxing

String output_str

Demonstration Program

PALINDROME.JAVA

PALINDROMEIGNORENONALPHANUMERIC.JAVA

Regular Expression

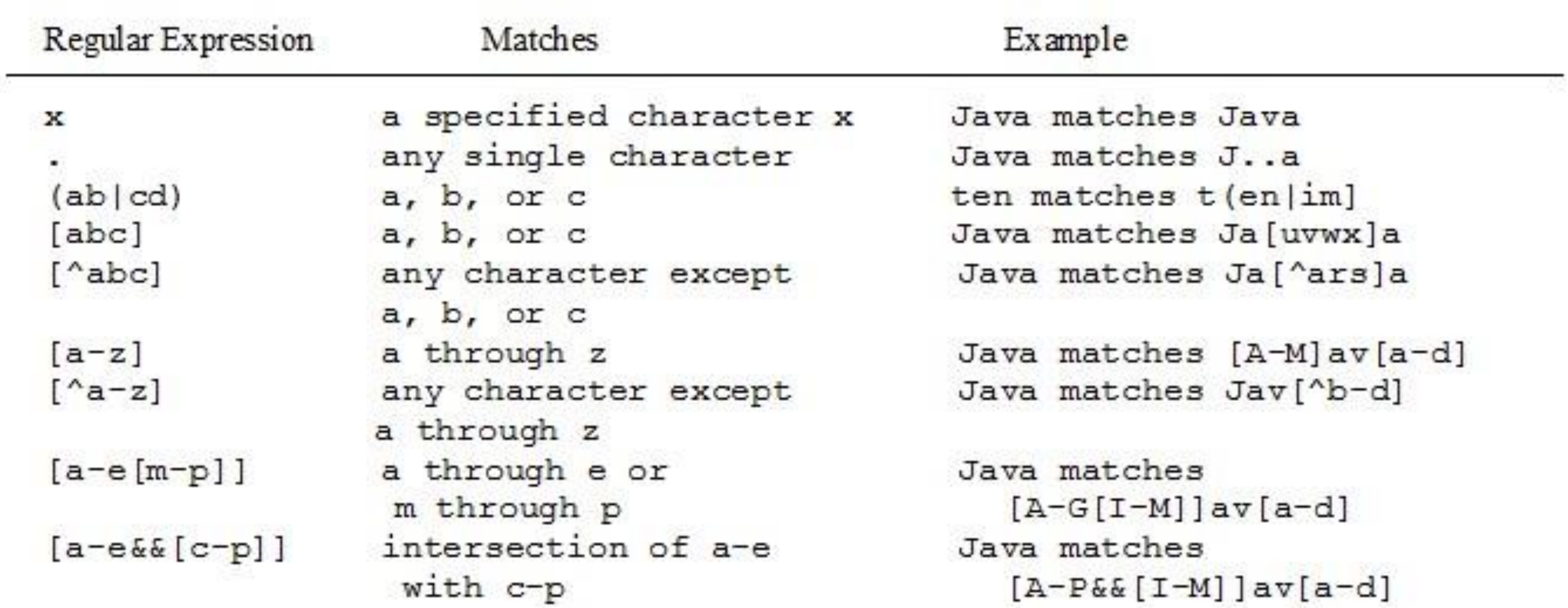
LECTURE 10

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.

Regular Expression Syntax

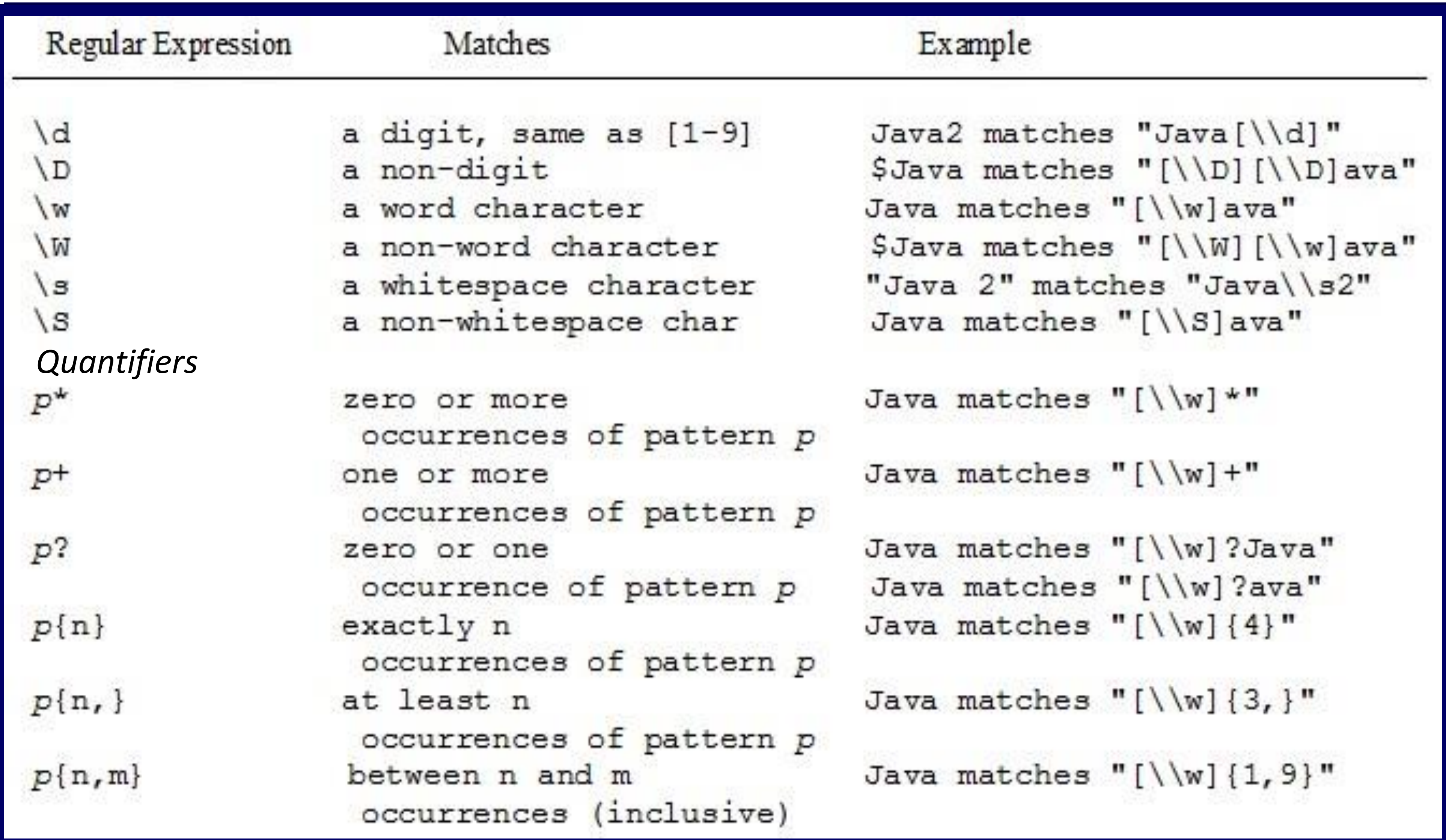
[or-set] [^not-set], Range: [a-z], Union: [set1[set2]] Intersection: [set1&&set2]



Regular Expression	Matches	Example
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab|cd)	a, b, or c	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]

Regular Expression Syntax

\one-wildcard-type-letter (in string \\d, first \ is escape)



Regular Expression	Matches	Example
\d	a digit, same as [1-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java matches "[\\w]ava"
\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"
Quantifiers		
*p**	zero or more occurrences of pattern *p*	Java matches "[\\w]*"
p+	one or more occurrences of pattern *p*	Java matches "[\\w]+"
p?	zero or one occurrence of pattern *p*	Java matches "[\\w]?Java" Java matches "[\\w]?ava"
p{n}	exactly n occurrences of pattern *p*	Java matches "[\\w]{4}"
p{n,}	at least n occurrences of pattern *p*	Java matches "[\\w]{3,}"
p{n,m}	between n and m occurrences (inclusive)	Java matches "[\\w]{1,9}"

Matching, Replacing and Splitting by Patterns

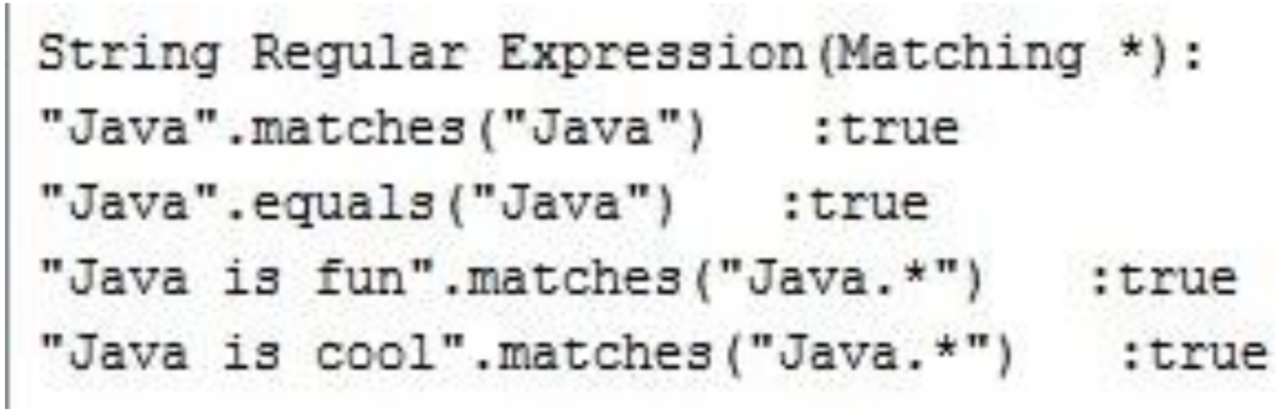
file extension, directory, student profile

- You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature, commonly known as *regular expression*. Regular expression is complex to beginning students. For this reason, two simple patterns are used in this section. Please refer to Supplement III.F, “Regular Expressions,” for further studies.

Matching, Replacing and Splitting by Patterns

file extension, directory, student profile

```
"Java".matches("Java");
"Java".equals("Java");
"Java is fun".matches("Java.*");
"Java is cool".matches("Java.*");
"Java is cool".matches("Java\.+");
```



```
String Regular Expression(Matching *):
"Java".matches("Java")    :true
"Java".equals("Java")    :true
"Java is fun".matches("Java.*")    :true
"Java is cool".matches("Java.*")    :true
```

This try to match for ‘.’ one or more periods, the result is false.

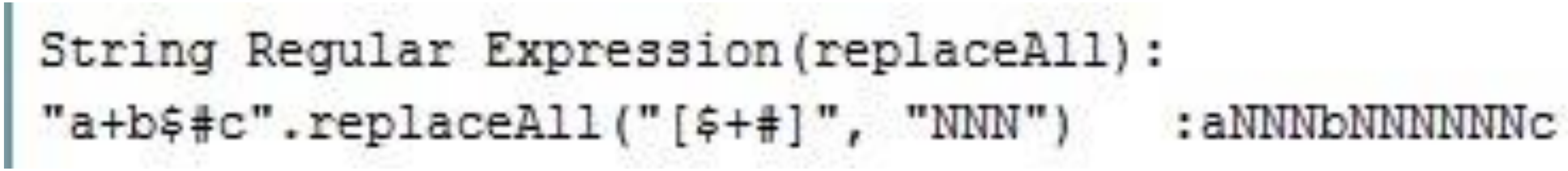
Matching, Replacing and Splitting by Patterns

- The replaceAll, replaceFirst, and split methods can be used with a regular expression. For example, the following statement returns a new string that replaces $, +, or # in "a+b$#c" by the string NNN.

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Matching, Replacing and Splitting by Patterns

- Here the regular expression [$+#] specifies a pattern that matches $, +, or #. So, the output is aNNNbNNNNNNc.

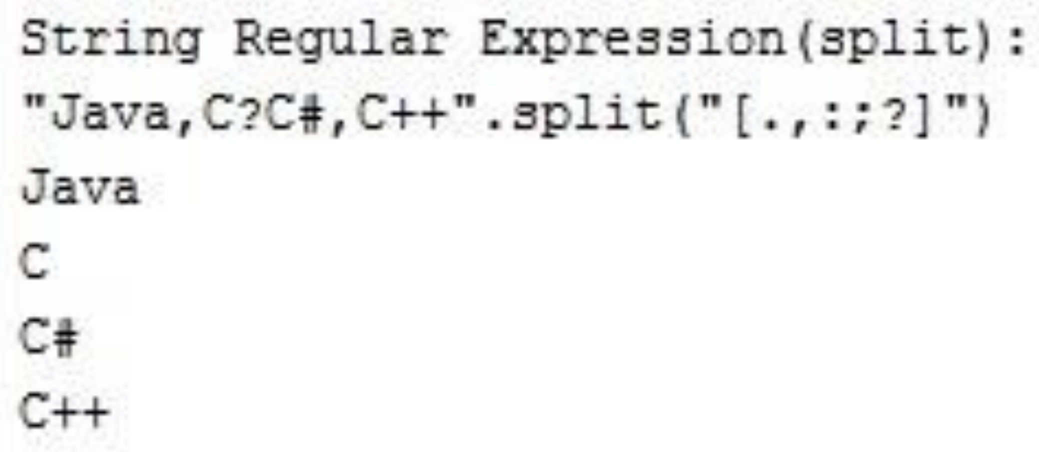


```
String Regular Expression(replaceAll):
"a+b$#c".replaceAll("[$+#]", "NNN")    :aNNNbNNNNNNc
```

Matching, Replacing and Splitting by Patterns

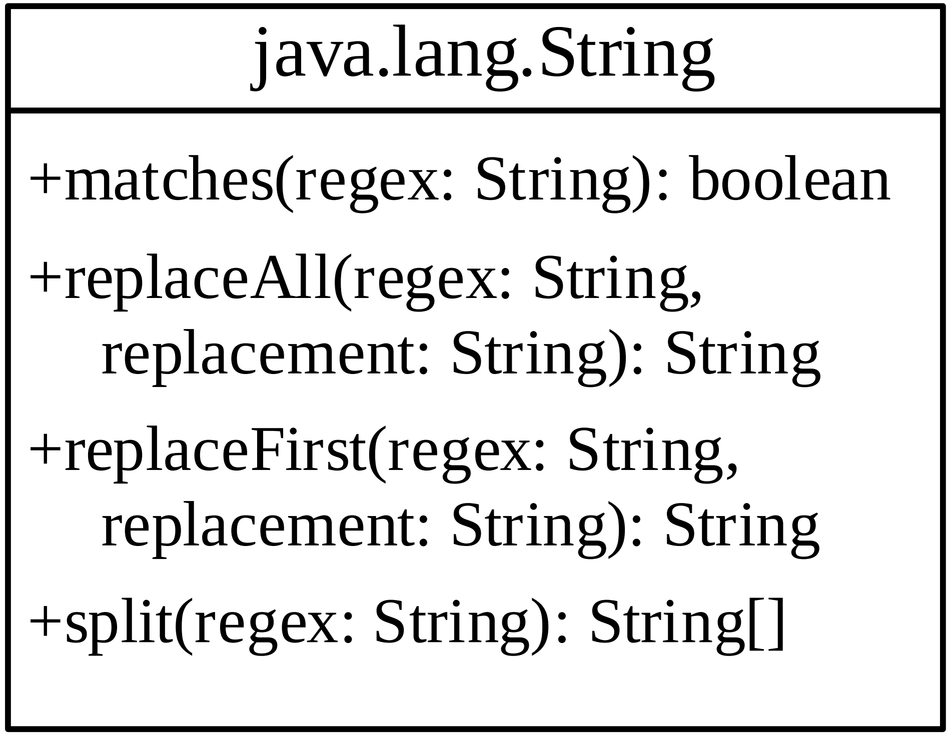
The following statement splits the string into an array of strings delimited by some punctuation marks.

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

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



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

Replacing and Splitting Strings



java.lang.String

+matches(regex: String): boolean

+replaceAll(regex: String, replacement: String): String

+replaceFirst(regex: String, replacement: String): String

+split(regex: String): String[]

Returns true if this string matches the pattern.

Returns a new string that replaces all matching substrings with the replacement.

Returns a new string that replaces the first matching substring with the replacement.

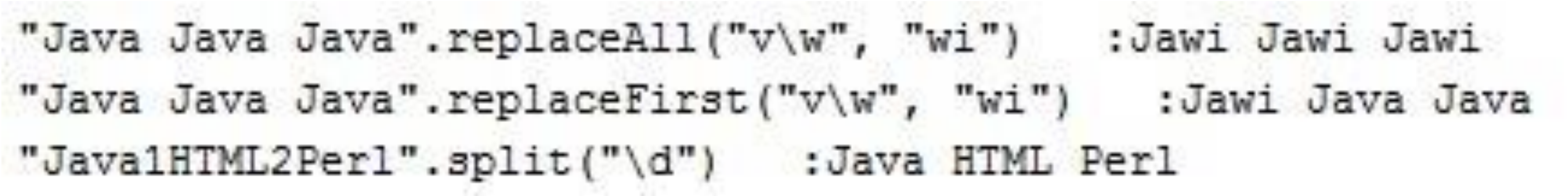
Returns an array of strings consisting of the substrings split by the matches.

Examples

String s = "Java Java Java".replaceAll("v\\w", "wi") ;

String s = "Java Java Java".replaceFirst("v\\w", "wi");

String[] s = "Java1HTML2Perl".split("\\d");



```
"Java Java Java".replaceAll("v\w", "wi")    :Jawi Jawi Jawi
"Java Java Java".replaceFirst("v\w", "wi")    :Jawi Java Java
"Java1HTML2Perl".split("\d")    :Java HTML Perl
```