String and Wrappers

Shing-Chi Cheung

Computer Science and Engineering

HKUST

Why support a built-in String class?

- Popularity – String processing is common
  - Reports/Financial Information/Documents in general
  - Information exchanges over Web
- String processing and I/O are incredibly important
- String class provides a uniform and efficient implementation to process strings

Constructing Strings

Syntax:

```
String <var> = new String(); // empty string
String <var> = new String(<string literal>);
String <var> = <string literal>; // shorthand initializer
```

Examples:

- **var** message = **new** String("Welcome to Java");
- **var** message = "Welcome to Java";
  - A shorthand initializer for string creation

Text Blocks (Multi-line Strings) - New

- Preview in Java 14; standard feature in Java 15

```
var html = """
        <html>
          <body>
            <p>Hello, world</p>
          </body>
        </html>
        """;
```

TextBlockTest

- html references the following String object:

```
"<html>\n    <body>\n        <p>Hello, world</p>\n    </body>\n</html>\n"
```

Strings Are Immutable

- A String object's contents are immutable (cannot be changed).

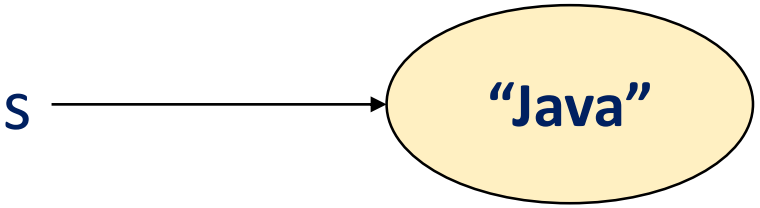
***Question**: Does the following code change the content of the String object from "Java" to HTML"?*

```
var s = "Java";
s = "HTML";
```

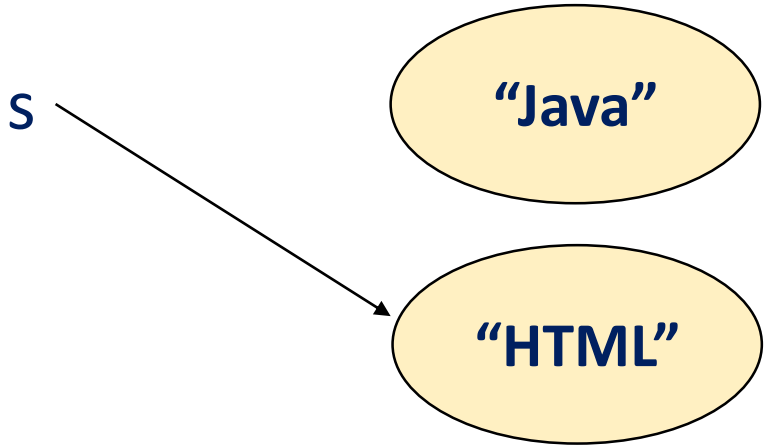
What happens?

```
var s = "Java";
s = "HTML";
```

After executing
var s = “Java”;



After executing
s = “HTML”;



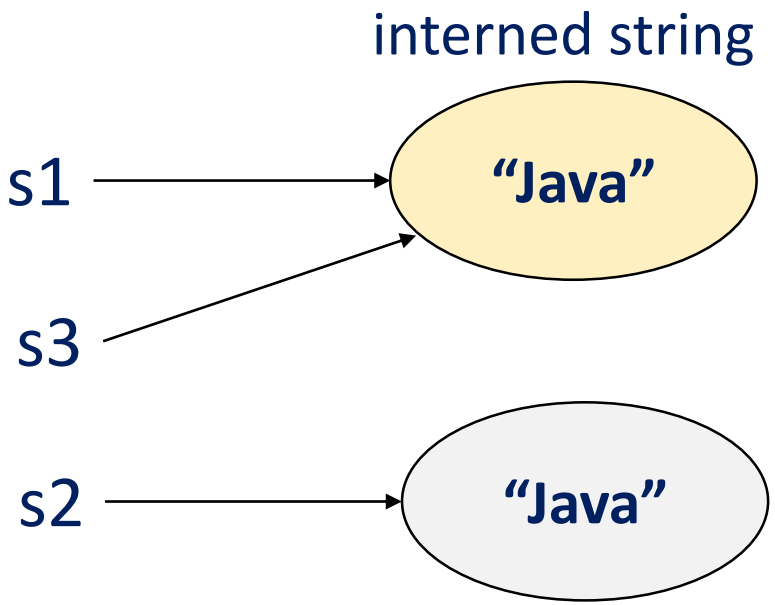
Interned Strings

- To improve efficiency and save memory, Java JVM uses a unique instance for string literals with the same character sequence.

```
var s1 = "Java"; // interned string
var s2 = new String("Java");
var s3 = "Java"; // interned string
System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

Console output:

```
s1 == s2 is false
s1 == s3 is true
```



InternedString.java

Interned Strings

- **Question**: How many objects are created by this code snippet?

```
var s1 = "Java"; // interned string
var s2 = new String("Java");
var s3 = "Java"; // interned string
System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

Commonly Used String Comparisons

- **boolean** equals(Object s) ☞ *Why not boolean equals(String s)?*
  - returns **true** if this and s have the same content; otherwise **false**
  - **boolean** equalsIgnoreCase(String s)
- **int** compareTo(String s)
  - compare two strings lexicographically
  - **boolean** compareToIgnoreCase(String s)
- **boolean** startsWith(String prefix)
- **boolean** endsWith(String suffix)

Within an instance method or a constructor, `this` is a reference to the *current object* — the object whose method or constructor is being called.
-- *from Oracle Java Documentation*

String Comparisons

- String class provides **boolean** equals(Object s)
- It checks whether this and s have the same contents
  - E.g., s1.equals(s2);
  - It overrides the equals(Object o) at the java.lang.Object class
  - java.lang.String extends java.lang.Object
  - Note: All java classes implicitly extend java.lang.Object
- Use operator == to check if the two operands reference the same object
  - E.g., s1 == s2

String Comparisons

```java
var s1 = new String("Welcome");
var s2 = "Welcome";

if (s1.equals(s2))
  System.out.println("s1 and s2 have the same contents");

if (s1 == s2)
  System.out.println("s1 and s2 have the same reference");

if (s1.compareTo(s2) > 0)
  System.out.println("s1 is greater than s2");
else if (s1.compareTo(s2) == 0)
  System.out.println("s1 and s2 have the same contents");
else
  System.out.println("s1 is less than s2");
```

Output:

```
s1 and s2 have the same contents
s1 and s2 have the same contents
```

StringComparison.java

java.lang.Object – an **important** Java class

- A Java class extends java.lang.Object unless otherwise specified. For example:

```
public class Person {
  …
}
```

=

```
import java.lang.*;
public class Person extends Object {
  …
}
```

- As a result, all Java classes (except java.lang.Object) are descended from java.lang.Object
- Instance methods defined in java.lang.Object are inherited by all Java classes

java.lang.Object – an **important** Java class

- Commonly used instance methods defined in java.lang.Object:
  - **public boolean** equals(Object obj) *// compare this with obj*
  - **public int** hashCode() *// returns a hash code of this*
  - **public** String toString() *// returns a string representation of this*
  - **protected** Object clone() *// returns a copy of this*
- They are normally overridden before use

```
class MyClass { // non-public class
  @Override
  public boolean equals(Object obj) { … } // public method
}
```

java.lang.Object – an **important** Java class

- Commonly used instance methods defined in java.lang.Object:
  - **public boolean** equals(Object obj) *// compare this with obj*
  - **public int** hashCode() *// returns a hash code of this*
  - **public** String toString() *// returns a string representation of this*
  - **protected** Object clone() *// returns a copy of this*
- They are normally overridden before use
- We will discuss the first three methods in the following slides and postpone the discussion of the clone method to a later topic

The equals(Object o) in java.lang.Object

- The equals(Object o) method compares two objects.
- The inherited implementation of equals(Object o) from the Object class compares if two variables share the same object references.

```
public boolean equals(Object obj) { // implementation in Object
  return (this == obj);
}
```

compare object references

A better practice is:
Override equals() in our defined Java class to compare contents instead of object references.

```
class Circle {
// …
  public boolean equals(Object o) {
    if (o instanceof Circle)
      return radius == ((Circle)o).radius;
    else
      return false;
} }
```

The equals(Object o) in java.lang.Object

- Pattern variables (new preview feature in Java 14)
  - *<Variable>* instanceof *<Class>* *<patternVariable>*
  - Example: if (obj *instanceof* Circle c) { System.out.println(c.radius); }

```
public boolean equals(Object obj) { // implementation in Object
  return (this == obj);
}
```

compare object references

A better practice is:
Override equals() in our defined Java class to compare contents instead of object references.

```
class Circle {
// ...
  public boolean equals(Object o) {
    return (o instanceof Circle c &&
      radius == c.radius);
  }
}
```

pattern variable

The hashcode() in java.lang.Object

- Default implementation of hashcode() returns a random integer unique to the current instance's memory address.
- According to Java documentation (below), we should override hashcode if we override equals.
  - *"If two objects are equal according to the equals(Object) method, then calling the hashcode() method on each of the two objects must produce the same integer result."*
- So, if equals is overridden to compare contents, hashcode should be overridden at the same time to hash the concerned contents.

The toString() method in java.lang.Object

- The default implementation returns a string consisting of:
  - a class name of which the object is an instance
  - the at sign (@), and
  - a number representing this object.

```
var p = new Person("Peter Wong", 19);
System.out.println(p.toString());
System.out.println(p); // invoke p.toString() implicitly
```

- The code outputs something like Person@f5f2bb7
- Usually, we override toString() to return meaningful information of the object

Person.java

String Length, Characters, and Combining Strings

java.lang.String	
+length(): int	Returns the number of characters in this string.
+charAt(index: int): char	Returns the character at the specified index from this string.
+concat(s1: String): String	Returns a new string that concatenate this string with string s1.

Finding String Length

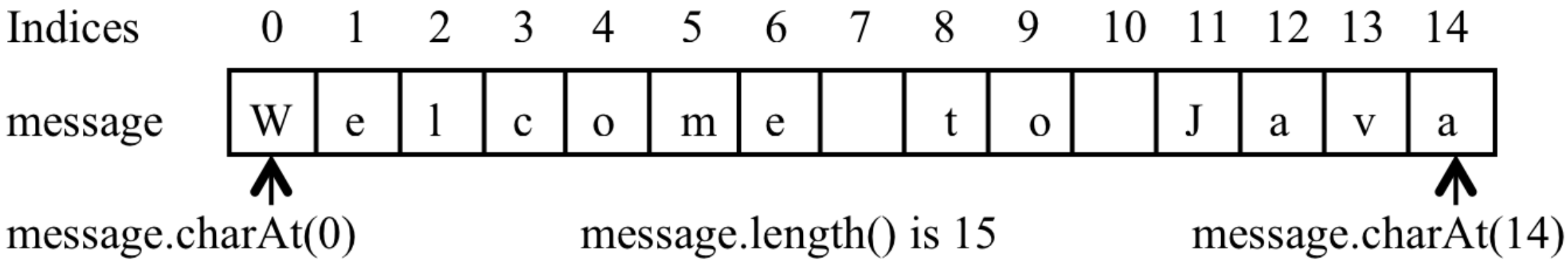
- Finding string length using the *length()* method:

```
message = "Welcome";
message.length() // returns 7
```

StringLength.java

Retrieving Individual Characters in a String

- Do not use message[0] *// String is not an array*
- Use message.charAt(index)
- Index starts from 0



CharacterExtraction.java

String Concatenation

```
var s3 = s1.concat(s2);
var s3 = s1 + s2;
```

- s1 + s2 + s3 + s4 + s5 is the same as:

  (((s1.concat(s2)).concat(s3)).concat(s4)).concat(s5);

StringConcatenation.java

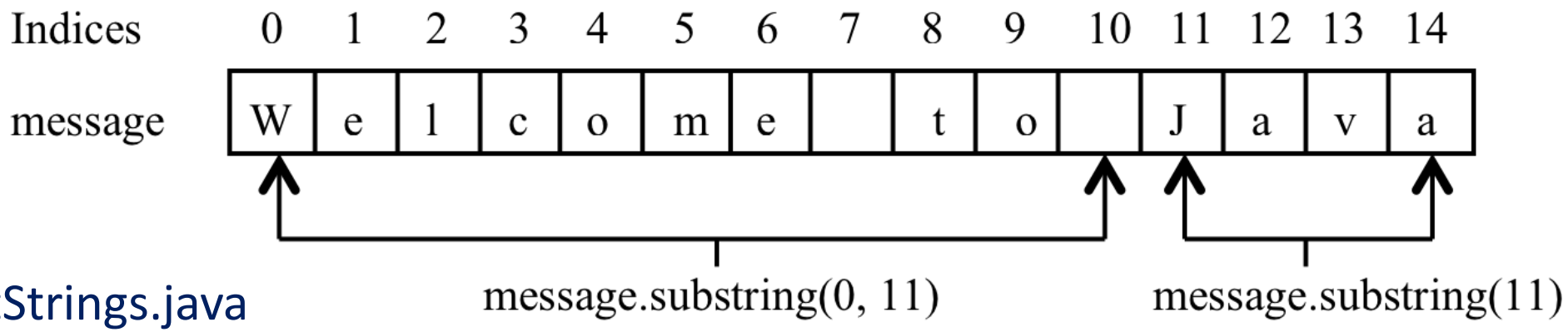
Extracting Substrings

java.lang.String	
+subString(beginIndex: int): String	Returns this string's substring that begins with the character at the specified beginIndex and extends to the end of the string.
+subString(beginIndex: int, endIndex: int): String	Returns this string's substring that begins at the specified beginIndex and extends to the character at index endIndex – 1. Note that the character at endIndex is not part of the substring.

Extracting Substrings

- You can extract a substring from a string using the substring method in the String class.

```
var s1 = "Welcome to Java";
var s2 = s1.substring(0, 11) + "HTML";
```



ExtractStrings.java

Converting, Replacing, and Splitting Strings

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.

ConvertReplaceSplitStrings.java

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

a String array

```
var tokens = "Java#HTML#Perl".split("#", 0);
for (var token: tokens)
  System.out.print(token + " ");
```

Match the pattern as many times as possible

Output:

```
Java HTML Perl
```

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.

FindCharSubString.java

Convert Character and Numbers to Strings

- The String class provides several static **valueOf** factory 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 “5.44”.

ConvertCharNumberToStrings.java

Convert Strings to Numbers

- Use **valueOf** factory methods to convert a String representation of number to the part, an array of characters, and numeric values to the actual number.
- Two ways:
  - through method valueOf to get an object
  - through method parseXX to get a primitive type value
- Example
  - **var** i = Integer.valueOf("999"); // return an Integer object with a value of 999
  - **var** i = Integer.parseInt("999"); // return an int value of 999
  - **var** l = Long.valueOf("99999999999"); // return a Long object with a value of 99…
  - **var** l = Long.parseLong("99999999999"); // return a long value of 99…
  - **var** l = Long.parseLong("I am not a number"); // error

more discussion later …

Problem: Finding Palindromes

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

CheckPalindrome.java

Use String.format() to format a string

- We can format a string like C++ printf.

```
var one = 20459954;
var two = 100597890.248907;
var s = String.format("The rank is %,d out of %,.2f", one, two);
System.out.println(s);
```

Output: The rank is 20,459,954 out of 100,597,890.25

FormatString.java

StringBuilder and StringBuffer

- The StringBuilder/StringBuffer class is an alternative to the String class.
- A StringBuilder/StringBuffer can be used wherever the String class is used.
- StringBuilder/StringBuffer is more flexible than String.
  - We 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.
  - StringBuffer is thread-safe, but StringBuilder is not.
  - Computation overhead: StringBuffer > StringBuilder > String

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.

PalindromeIgnoreNonAlphanumeric.java

Can we call and pass arguments to main()?

- Yes. We can call main() as a regular static method.
- For example, the main method in class B is invoked by a method in A, as shown below:

```
public class A {
  public static void main(String[] args) {
    String[] strings = {"New York",
      "Boston", "Atlanta"};
    B.main(strings);
  }
}
```

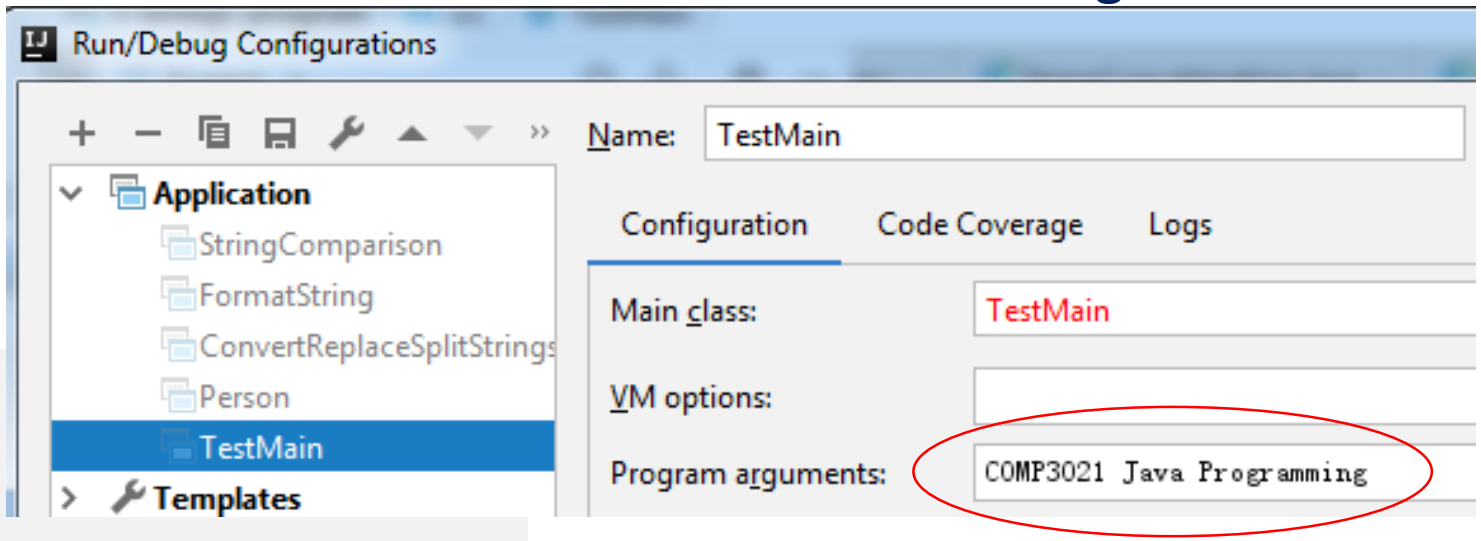
```
class B {
  public static void main(String[] args) {
    for (var s: args)
      System.out.println(s);
  }
}
```

A.java

Command-Line Parameters

```
class TestMain {
  public static void main(String[] args) {
  ...
  }
}
```

IntelliJ: Run -> Edit Configurations…



line commands:

javac TestMain.java

java TestMain COMP3021 Java Programming

args[0] *args[1]* *args[2]*

TestMain.java

Problem: Calculator

- Objective: Write a program that will perform binary operations on integers. The program receives three parameters: an operator and two integers.

```
java --enable-preview Calculator "2 + 3"
java --enable-preview Calculator "2 - 3"
java --enable-preview Calculator "2 / 3"
java --enable-preview Calculator "2 * 3"
```

Calculator.java

WRAPPER CLASSES FOR PRIMITIVE TYPES

Primitive Type	Wrapper Class
int	Integer
double	Double
boolean	Boolean
…	…

Can we write pure OO Java programs?

- Yes, Java has a built-in wrapper class for each primitive type such as int, double, long, float, bool, char, …
  - E.g.,

```
var intObj = new Integer(0); // create an Integer object
intObj++;
```

- In principle, we can write an OO program without using primitive type variables

Wrapper Classes

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Numeric Wrapper Class Constructors

- We can construct a wrapper object either from a primitive type value or from a string representing the numeric value.

- The constructors for Integer and Double are:

```
public Integer(int value)
public Integer(String s)
public Double(double value)
public Double(String s)
```

Numeric Wrapper Class Constants

- Each numerical wrapper class defines MAX_VALUE and MIN_VALUE
- MAX_VALUE gives the maximum value of the corresponding primitive type
- For Byte, Short, Integer, and Long, MIN_VALUE gives the minimum byte, short, int, and long values. For Float and Double, MIN_VALUE gives the minimum *positive* float and double values. Examples:
  - The max integer: 2,147,483,947
  - The min positive float: 1.4e-45
  - The max double floating-point number: 1.79799313489231570e+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" numeric objects into their primitive type values. For example:

```
var three = new Integer(3).intValue();
var value = new Integer(3).doubleValue();
var pi = new Double(3.14159).doubleValue();
```

The Static valueOf Factory Methods

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

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

var integerObject = Integer.valueOf("12");
```

Integer.valueOf() vs new Integer()

public static Integer *valueOf*(int i)

Returns a Integer instance representing the specified int value. If a new Integer instance is not required, this method should generally be used in preference to the constructor Integer(int), as this method is likely to yield significantly better space and time performance by caching frequently requested values.

Parameters:
i - an int value.
Returns:
a Integer instance representing i.

Note: Integer objects created by valueOf() are annotated with @HotSpotIntrinsicCandidate, notifying HotSpot VM that it can use optimised assembly level code to gain better performance.

Integer.valueOf() vs new Integer()

```
public static void main(String[] args) {

  var a = new Integer(1);
  var b = new Integer(1);
  System.out.println("a==b? " + (a==b));

  var c = Integer.valueOf(1);
  var d = Integer.valueOf(1);
  System.out.println("c==d? " + (c==d));
}
```

Output:

a==b? false
c==d? true

- The use of **new** Integer(**int**) has been deprecated since Java 9.
- We are advised to use Integer.*valueOf*(**int**) instead.

The Methods for Parsing Strings into Numbers

- We use 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 a parsing methods to parse a numeric string into its numeric value.

```
var pi = Double.parseDouble("3.14159");
```

Automatic Conversion Between Primitive Types and Wrapper Class Types

Java allows a primitive type and its wrapper class to be converted automatically.

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

(a)

Equivalent to

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

auto-boxing

(b)

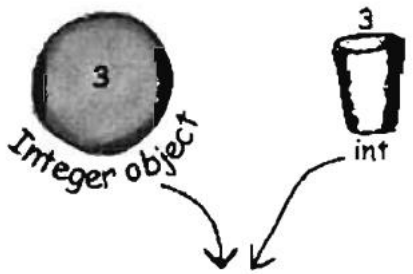
```
Integer[] intArray = {2, 4, 3};  // auto-boxing 3 Integer objects using valueOf()
System.out.println(intArray[0] + intArray[1] + intArray[2]);  // unboxing
```

Unboxing

More on Autoboxing …

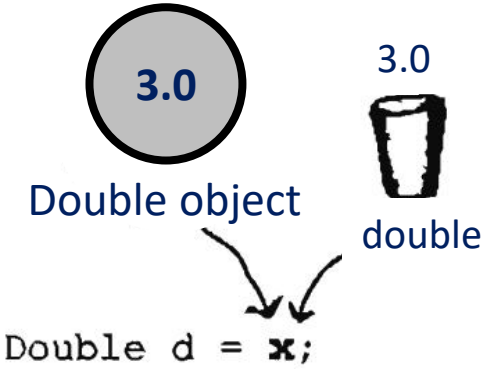
extracted from Head First Java

- Method parameters

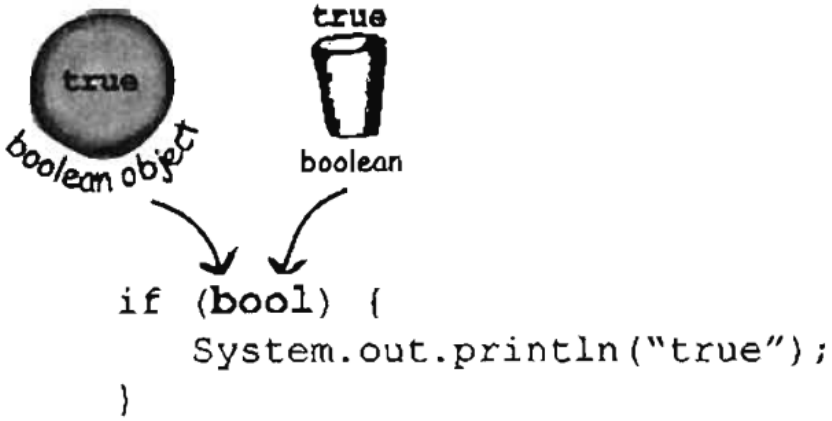



```
void takeNumber(Integer i) { }
```

- Assignments



- Boolean expressions



- Return values

```
int giveNumber() {
  return x;
}
```