Wrapper Classes

Paul Fodor

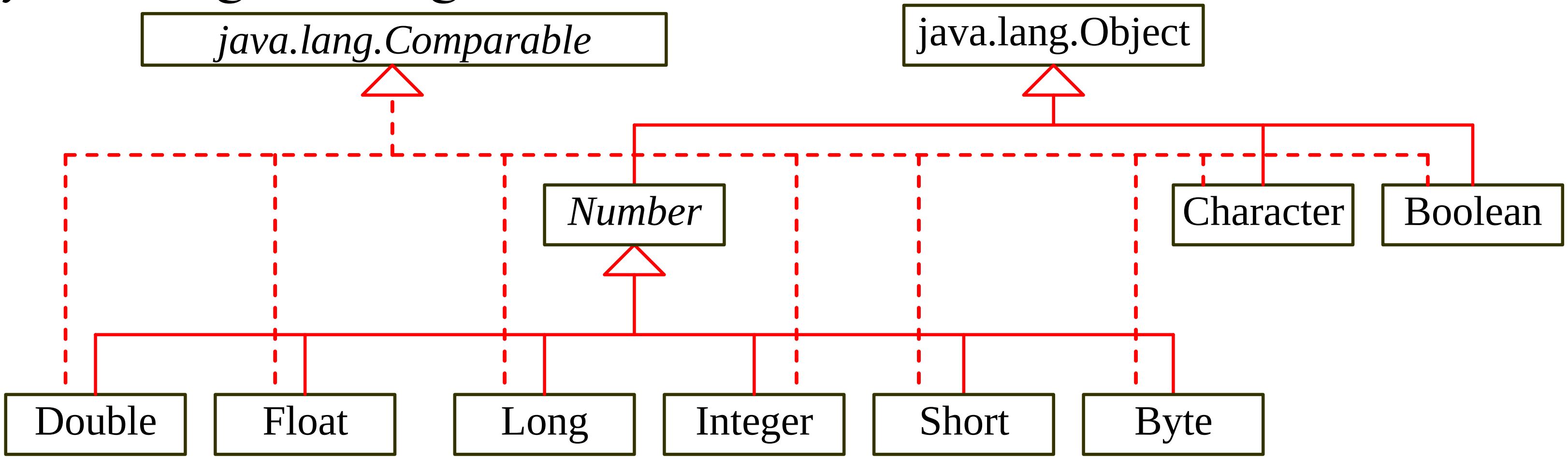
CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

Wrapper Classes

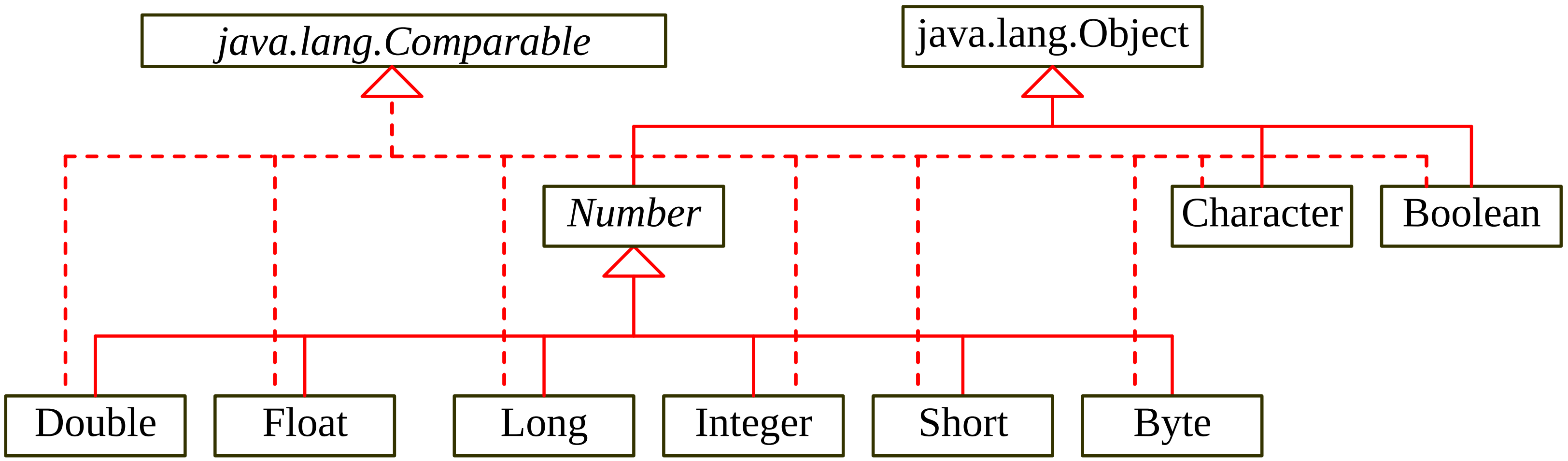
- Primitive data types in Java ➔ **Better performance**
  - However, data structures (ArrayList) expect objects as elements
- Each primitive type has a wrapper class: Boolean, Character, Short, Byte, Integer, Long, Float, Double



  - The wrapper classes do not have no-arg constructors
  - The instances of all wrapper classes are immutable: their internal values cannot be changed once the objects are created

Wrapper Classes

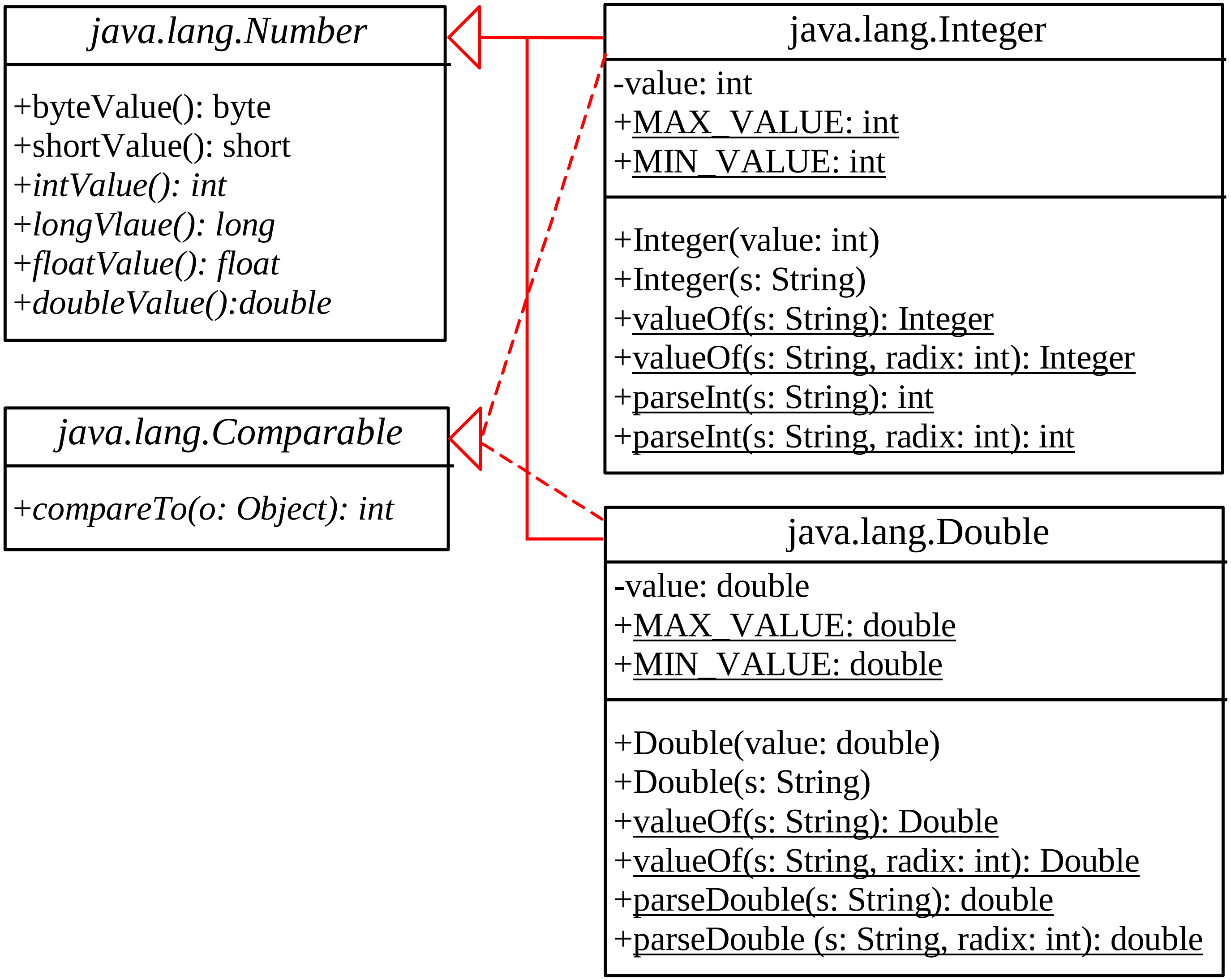
- Each wrapper class overrides the **`toString`** and **`equals`** methods defined in the **`Object`** class
- Since these classes implement the **`Comparable`** interface, the **`compareTo`** method is also implemented in these classes



The ***Number*** Class

- Each numeric wrapper class extends the abstract ***Number*** class:
  - The abstract ***Number*** class contains the methods ***doubleValue***, ***floatValue***, ***intValue***, ***longValue***, ***shortValue***, and ***byteValue*** to "convert" objects into primitive type values
  - The methods ***doubleValue***, ***floatValue***, ***intValue***, ***longValue*** are ***abstract***
    - The methods ***byteValue*** and ***shortValue*** are not ***abstract***, which simply return ***(byte)intValue()*** and ***(short)intValue()***, respectively
    - Each numeric wrapper class implements the abstract methods ***doubleValue***, ***floatValue***, ***intValue*** and ***longValue***

The **Integer** and **Double** Classes



Wrapper Classes

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

- The constructors for **`Integer`** and **`Double`** are:

```
public Integer(int value)

public Integer(String s)

public Double(double value)

public Double(String s)
```

Numeric Wrapper Class Constants

- Each numerical wrapper class has the constants **MAX_VALUE** and **MIN_VALUE**:
  - **MAX_VALUE** represents the maximum value of the corresponding primitive data type
  - For **Float** and **Double**, **MIN_VALUE** represents the minimum *positive* **float** and **double** values
  - The maximum integer: 2,147,483,647
  - The minimum positive float: 1.4E-45
  - The maximum double floating-point number: 1.79769313486231570e+308d

The `static` `valueOf` methods

- The numeric wrapper classes have a **`static`** method **`valueOf(String s)`** to create a new object initialized to the value represented by the specified string:

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

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

```
double d = Double.parseDouble("12.4");
int i = Integer.parseInt("12");
```

Wrapper Classes

- Automatic Conversion Between Primitive Types and Wrapper Class Types:
  - Since JDK 1.5, Java allows primitive type and wrapper classes to be converted automatically:
    - ***boxing*** of primitive types into wrapper types when objects are needed

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

Equivalent

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

- ***unboxing*** of wrapper types into primitive types when primitive types are needed

```
int n = intArray[0] + intArray[1] + intArray[2];
```

Unboxing

Arrays are objects

- Arrays are objects:
  - An array is an instance of the **`Object`** class

```
new int[10] instanceof Object                    true
```

  - If **`A`** is a subclass of **`B`**, every instance of **`A[]`** is an instance of **`B[]`**

```
new GregorianCalendar[10] instanceof Calendar[] true
new Calendar[10] instanceof Object[]             true
new Calendar[10] instanceof Object               true
```

- Although an **`int`** value can be assigned to a **`double`** type variable, **`int[]`** and **`double[]`** are two incompatible types because they are not classes:
  - We cannot assign an **`int[]`** array to a variable of **`double[]`** array: compiler error: **`double[] a = new int[10];`**

Sorting an Array of Objects

- Java provides a **`static`** **`sort`** method for sorting an array of **`Object`** in the **`java.util.Arrays`** class that uses the **`Comparable`** interface:

```
java.util.Arrays.sort(intArray);
```

Sorting an Array of Objects

```
public class GenericSort {
  public static void main(String[] args) {
    Integer[] intArray={new Integer(2),new Integer(4),new Integer(3)};
    sort(intArray); // or Arrays.sort(intArray);
    printList(intArray);
  }
  public static void sort(Object[] list) {
    Object currentMax;
    int currentMaxIndex;
    for (int i = list.length - 1; i >= 1; i--) {
      currentMax = list[i];
      currentMaxIndex = i; // Find the maximum in the list[0..i]
      for (int j = i - 1; j >= 0; j--) {
        if (((Comparable)currentMax).compareTo(list[j]) < 0) {
          currentMax = list[j];
          currentMaxIndex = j;
        }
      }
      list[currentMaxIndex] = list[i];
      list[i] = currentMax;
    }
  }
  public static void printList(Object[] list) {
    for (int i=0;i<list.length;i++) System.out.print(list[i]+" ");}}
```

The objects are instances of the Comparable interface and they are compared using the compareTo method.

`BigInteger` and `BigDecimal`

- **`BigInteger`** and **`BigDecimal`** classes in the **`java.math`** package:
  - For computing with very large integers or high precision floating-point values
    - **`BigInteger`** can represent an integer of any size
    - **`BigDecimal`** has no limit for the precision (as long as it's finite=terminates)
  - Both are *immutable*
  - Both extend the ***`Number`*** class and implement the **`Comparable`** interface.

BigInteger and BigDecimal

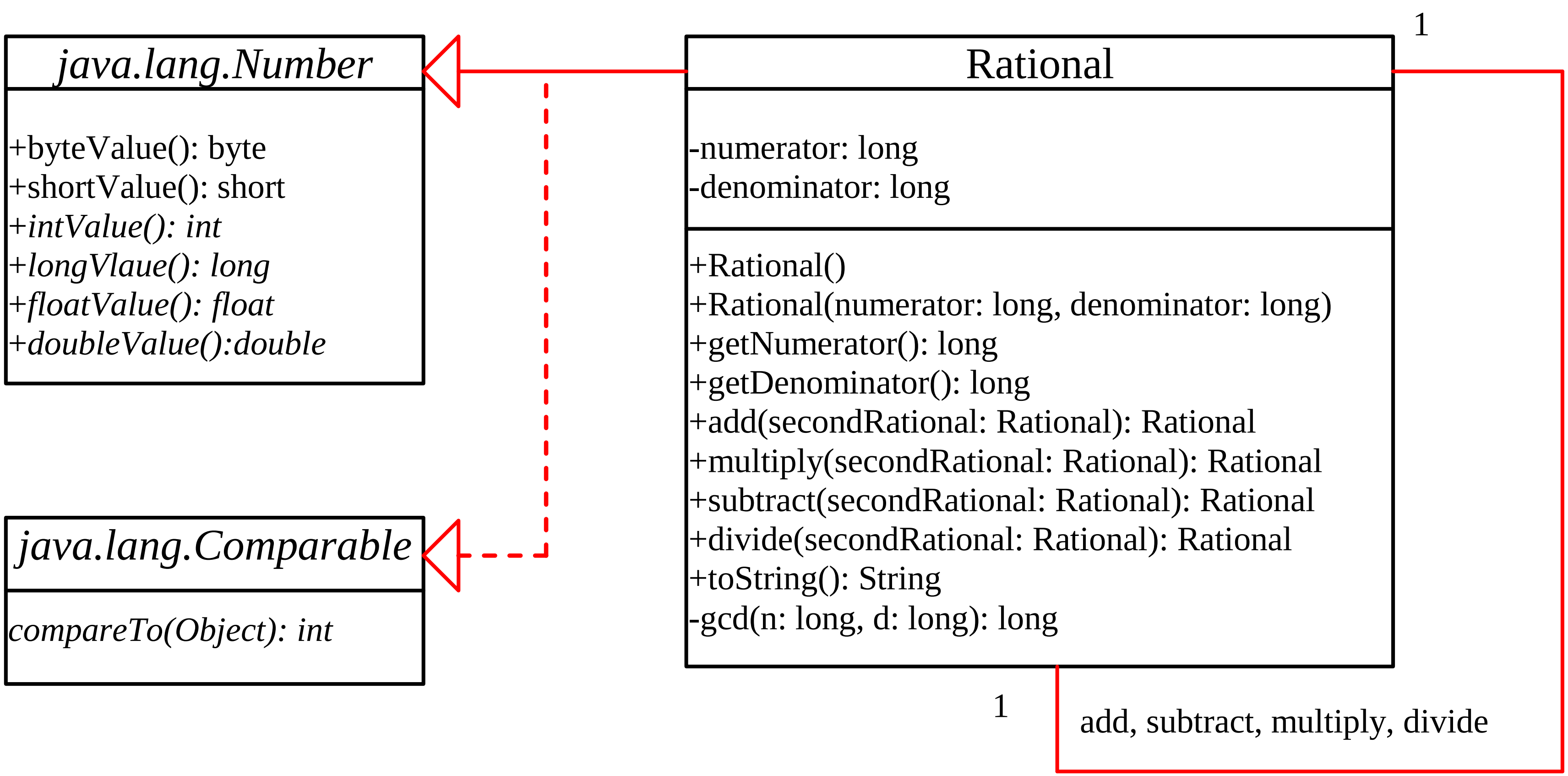
```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
                    18446744073709551614
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
                    0.33333333333333333334
```

BigInteger and BigDecimal

```
import java.math.*;
public class LargeFactorial {
  public static void main(String[] args) {
    System.out.println("50! is \n" + factorial(50));
  }
  public static BigInteger factorial(long n) {
    BigInteger result = BigInteger.ONE;
    for (int i = 1; i <= n; i++)
      result = result.multiply(new BigInteger(i+""));
    return result;
  }
}
```

30414093201713378043612608166064768844377641
568960512000000000000

Case Study: The `Rational` Class



```
public class Rational extends Number implements Comparable {
  private long numerator = 0;
  private long denominator = 1;
  public Rational() { this(0, 1); }
  public Rational(long numerator, long denominator) {
    long gcd = gcd(numerator, denominator);
    this.numerator = ((denominator > 0) ? 1 : -1) * numerator / gcd;
    this.denominator = Math.abs(denominator) / gcd;
  }
  private static long gcd(long n, long d) {
    long n1 = Math.abs(n);
    long n2 = Math.abs(d);
    int gcd = 1;
    for (int k = 1; k <= n1 && k <= n2; k++) {
      if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
    }
    return gcd;
  }
  public Rational add(Rational secondRational) {
    long n = numerator * secondRational.getDenominator() +
      denominator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return new Rational(n, d);
  }
```

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

```
  public Rational subtract(Rational secondRational) {
    ...                                // or implement inverse and use add method
  }
```

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

$$-\left(\frac{a}{b}\right) = \frac{-a}{b}$$

```
  // multiply, divide

  /** Override the abstract intValue method in java.lang.Number */
  public int intValue() {   return (int)doubleValue();   }
  public double doubleValue() {
    return ((double)numerator)/denominator;
  }
  // ... Override all the abstract *Value methods in java.lang.Number

  /** Override the compareTo method in java.lang.Comparable */
  public int compareTo(Object o) {
    if ((this.subtract((Rational)o)).getNumerator() > 0) return 1;
    else if ((this.subtract((Rational)o)).getNumerator()<0) return -1;
    else return 0;
  }
  public static void main(String[] args) {
    Rational r1 = new Rational(4, 2);
    Rational r2 = new Rational(2, 3);
    System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
  }
}
```

```
public class SomethingCloneable implements Cloneable {
  public boolean equals(Object o){
    return true;
  }
  public static void main(String[] args)
      throws CloneNotSupportedException {
    SomethingCloneable s1 = new SomethingCloneable();
    SomethingCloneable s2 = (SomethingCloneable) s1.clone();
    System.out.println("s1 == s2 is " + (s1 == s2));
       // false
    System.out.println("s1.equals(s2) is " + s1.equals(s2));
       // true
  }
}
```