OOP with Java

29. Autoboxing

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

1. Introduction
2. Wrapper Types
3. Autoboxing
4. Unintended Consequences
5. Summary

website

Introduction

- In Java, we can distinguish two kinds of types

Introduction

- In Java, we can distinguish two kinds of types:
  1. the primitive types `byte`, `short`, `int`, `long`, `boolean`, `char`, `float`, and `double`

Introduction

- In Java, we can distinguish two kinds of types: primitive and objects:
  1. the primitive types `byte`, `short`, `int`, `long`, `boolean`, `char`, `float`, and `double`
  2. object types, basically everything else, including `Object`, `String`, arrays, and our own classes

Introduction

- In Java, we can distinguish two kinds of types: primitive and objects
- However, we can do something like this:

Listing: Putting Primitive Values into an Object Array

```java
/** Putting primitive types into an object array */
public class PrimitiveTypesInArray {
  /** The main routine
   * @param args we ignore it */
  public static void main(String[] args) {
    Object[] array = {1, true, 2.5d, 9f, 'x'};

    for(Object o: array) {
      System.out.println(o);
    } // prints 1\n true\n 2.5\n 9.0\n x
  }
}
```

Introduction

- In Java, we can distinguish two kinds of types: primitive and objects
- Or this:

Listing: Putting Primitive Values into an Object ArrayList

```java
import java.util.ArrayList;

/** Putting primitive types into a list */
public class PrimitiveTypesInList {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    ArrayList<Object> list = new ArrayList<>();

    list.add(1);
    list.add(true);
    list.add(2.5d);
    list.add(9f);
    list.add('x');
    System.out.println(list); // prints [1, true, 2.5, 9.0, x]
  }
}
```

Introduction

- In Java, we can distinguish two kinds of types: primitive and objects
- What's going on?

Wrapper Types

- For each primitive type, there exists one wrapper type

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	$\longrightarrow$	wrapper type	getter

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`
`int`	⟶	`java.lang.Integer`	`intValue()`

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`
`int`	⟶	`java.lang.Integer`	`intValue()`
`long`	⟶	`java.lang.Long`	`longValue()`

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`
`int`	⟶	`java.lang.Integer`	`intValue()`
`long`	⟶	`java.lang.Long`	`longValue()`
`boolean`	⟶	`java.lang.Boolean`	`booleanValue()`

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`
`int`	⟶	`java.lang.Integer`	`intValue()`
`long`	⟶	`java.lang.Long`	`longValue()`
`boolean`	⟶	`java.lang.Boolean`	`booleanValue()`
`float`	⟶	`java.lang.Float`	`floatValue()`

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`
`int`	⟶	`java.lang.Integer`	`intValue()`
`long`	⟶	`java.lang.Long`	`longValue()`
`boolean`	⟶	`java.lang.Boolean`	`booleanValue()`
`float`	⟶	`java.lang.Float`	`floatValue()`
`double`	⟶	`java.lang.Double`	`doubleValue()`

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`
`int`	⟶	`java.lang.Integer`	`intValue()`
`long`	⟶	`java.lang.Long`	`longValue()`
`boolean`	⟶	`java.lang.Boolean`	`booleanValue()`
`float`	⟶	`java.lang.Float`	`floatValue()`
`double`	⟶	`java.lang.Double`	`doubleValue()`
`char`	⟶	`java.lang.Character`	`charValue()`

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`
`int`	⟶	`java.lang.Integer`	`intValue()`
`long`	⟶	`java.lang.Long`	`longValue()`
`boolean`	⟶	`java.lang.Boolean`	`booleanValue()`
`float`	⟶	`java.lang.Float`	`floatValue()`
`double`	⟶	`java.lang.Double`	`doubleValue()`
`char`	⟶	`java.lang.Character`	`charValue()`

- These wrapper classes provide several useful `public static final` constants and utility methods for dealing with these types

Wrapper Types

- For each primitive type, there exists one wrapper type:

primitive type	⟶	wrapper type	getter
`byte`	⟶	`java.lang.Byte`	`byteValue()`
`short`	⟶	`java.lang.Short`	`shortValue()`
`int`	⟶	`java.lang.Integer`	`intValue()`
`long`	⟶	`java.lang.Long`	`longValue()`
`boolean`	⟶	`java.lang.Boolean`	`booleanValue()`
`float`	⟶	`java.lang.Float`	`floatValue()`
`double`	⟶	`java.lang.Double`	`doubleValue()`
`char`	⟶	`java.lang.Character`	`charValue()`

- These wrapper classes provide several useful `public static final` constants and utility methods for dealing with these types
- One instance of the wrapper class holds exactly one value of the primitive type

Autoboxing

- Whenever Java encounters an instance of a primitive type where an object is expected, it will wrap this instance into an instance of the corresponding wrapper class

Autoboxing

- Whenever Java encounters an instance of a primitive type where an object is expected, it will wrap this instance into an instance of the corresponding wrapper class
- Whenever Java encounters an instance of the wrapper class where the corresponding primitive type is needed, it will use the getter to extract the primitive type

Autoboxing

- Whenever Java encounters an instance of a primitive type where an object is expected, it will wrap this instance into an instance of the corresponding wrapper class
- Whenever Java encounters an instance of the wrapper class where the corresponding primitive type is needed, it will use the getter to extract the primitive type
- This is called autoboxing and auto-unboxing

Autoboxing

- This is called autoboxing

Listing: Autoboxing: Wrapper Objects in Array

```java
/** Putting primitive types into an object array */
public class PrimitiveTypesInArrayClasses {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    Object[] array = {1, true, 2.5d, 9f, 'x'};

    for(Object o: array) {
      System.out.println(o.getClass());
    } // prints class java.lang.Integer\n class java.lang.Boolean
// class java.lang.Double \n class java.lang.Float  \n class java.lang.Character
  }
}
```

`x.getClass()` returns an object representing the class of `x`, and that `toString()` method of this object returns the class name

Autoboxing

- This is called autoboxing

Listing: Autoboxing: Primitive in Object Array – What's actually going on?

```java
/** Putting primitive types into an object array */
public class PrimitiveTypesInArrayActual {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    Object[] array = {Integer.valueOf(1), Boolean.valueOf(true),
                      Double.valueOf(2.5d), Float.valueOf(9f),
                      Character.valueOf('x')};

    for(Object o: array) {
      System.out.println(o);
    } // prints 1\n true\n 2.5\n 9.0\n x
  }
}
```

Autoboxing

- This is called autoboxing

Listing: Autoboxing: Wrapper Objecta in ArrayList

```java
import java.util.ArrayList;

/** Putting primitive types into a list */
public class PrimitiveTypesInListClasses {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    ArrayList<Object> list = new ArrayList<>();

    list.add(1);
    list.add(true);
    list.add(2.5d);
    list.add(9f);
    list.add('x');
    for(Object o: list) {
      System.out.println(o.getClass());
    } // prints class java.lang.Integer\n class java.lang.Boolean
 // class java.lang.Double \n class java.lang.Float  \n class java.lang.Character
  }
}
```

`x.getClass()` returns an object representing the class of `x`, and that `toString()` method of this object returns the class name

Autoboxing

- This is called autoboxing

Listing: Autoboxing: Primitive in ArrayList – What’s actually going on?

```java
import java.util.ArrayList;

/** Putting primitive types into a list */
public class PrimitiveTypesInListActual {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    ArrayList<Object> list = new ArrayList<>();

    list.add(Integer.valueOf(1));
    list.add(Boolean.valueOf(true));
    list.add(Double.valueOf(2.5d));
    list.add(Float.valueOf(9f));
    list.add(Character.valueOf('x'));
    System.out.println(list); // prints [1, true, 2.5, 9.0, x]
  }
}
```

Unintended Consequence: Memory and Speed

- A Java object will take more memory than a primitive type

Unintended Consequence: Memory and Speed

- A Java object will take more memory than a primitive type
- For each wrapper object, there is the wrapped primitive type, plus 12 or-so bytes of management data (a pointer to the objects class, and data for the garbage collection, . . .)

Unintended Consequence: Memory and Speed

- A Java object will take more memory than a primitive type
- For each wrapper object, there is the wrapped primitive type, plus 12 or-so bytes of management data (a pointer to the objects class, and data for the garbage collection, . . .)
- In other words, wrapped primitive types are much bigger than plain primitive types

Unintended Consequence: Memory and Speed

- A Java object will take more memory than a primitive type
- For each wrapper object, there is the wrapped primitive type, plus 12 or-so bytes of management data (a pointer to the objects class, and data for the garbage collection, . . .)
- In other words, wrapped primitive types are much bigger than plain primitive types
- They are also much slower, due to wrapping and unwrapping

Unintended Consequence: Memory and Speed

- A Java object will take more memory than a primitive type
- For each wrapper object, there is the wrapped primitive type, plus 12 or-so bytes of management data (a pointer to the objects class, and data for the garbage collection, . . .)
- In other words, wrapped primitive types are much bigger than plain primitive types
- They are also much slower, due to wrapping and unwrapping
- This kicks especially in when you plug several primitive and object-based actions together in a loop

Unexpected Inquality

- The wrapper class instance wrapping a primitive type is an object

Unexpected Inquality

- The wrapper class instance wrapping a primitive type is an object:
  - the `==` and `!=` operators apply to the object instances, i.e., check for reference equality/inequality

Unexpected Inquality

- The wrapper class instance wrapping a primitive type is an object:
  - the `==` and `!=` operators apply to the object instances, i.e., check for reference equality/inequality
  - `<=`, `<`, `>=`, and `>` cannot be applied to references and hence force unboxing for numerical wrappers

Unexpected Inquality

- The wrapper class instance wrapping a primitive type is an object:
    - the `==` and `!=` operators apply to the object instances, i.e., check for reference equality/inequality
    - `<=`, `<`, `>=`, and `>` cannot be applied to references and hence force unboxing for numerical wrappers
- Now mix this with the fact that we cannot guarantee that `Integer.valueOf(1000)` will return the same object if called twice (*it will not!*)

Unexpected Inquality

- The wrapper class instance wrapping a primitive type is an object:
  - the `==` and `!=` operators apply to the object instances, i.e., check for reference equality/inequality
  - `<=`, `<`, `>=`, and `>` cannot be applied to references and hence force unboxing for numerical wrappers
- Now mix this with the fact that we cannot guarantee that `Integer.valueOf(1000)` will return the same object if called twice (*it will not!*)
- What do we get?

Checking Boxed Integer Equality

Listing: Checking Boxed Integer Equality

```
/** The same primitive value not necessarily wraps to the same object */
public class IntNotEqualsInt {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    int x, y;
    Integer a, b;

    x = 1000;
    y = 1000;
    System.out.println(x == y); // true

    a = x;
    b = y;
    System.out.println(a == b); // false

    x = a;
    y = b;
    System.out.println(x == y); // true
  }
}
```

Comparisons on Boxed Integers

Listing: Comparisons on Boxed Integers

```java
/** The same primitive value not necessarily wraps to the same object */
public class IntArghh {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    Integer a, b;

    a = 100;
    b = 100;
    System.out.println(a == b); // true : compares references, but 100<127..
    System.out.println(a < b);  // false: compares unboxed values
    System.out.println(a <= b); // true : compares unboxed values
    System.out.println(a > b);  // false: compares unboxed values
    System.out.println(a >= b); // true : compares unboxed values
    System.out.println(a != b); // false: compares references, but 100<127..
  }
}
```

Checking Boxed Doubles Equality

Listing: Checking Boxed Doubles Equality

```java
/** The same primitive value not necessarily wraps to the same object */
public class DoubleNotEqualsDouble {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    double x, y;
    Double a, b;

    x = 0d;
    y = 0d;
    System.out.println(x == y); // true

    a = x;
    b = y;
    System.out.println(a == b); // false

    x = a;
    y = b;
    System.out.println(x == y); // true
  }
}
```

Comparisons on Boxed Doubles

Listing: Comparisons on Boxed Doubles

```java
/** The same primitive value not necessarily wraps to the same object */
public class DoubleArghh {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    Double a, b;

    a = 0d;
    b = 0d;
    System.out.println(a == b); // false: compares references
    System.out.println(a < b);  // false: compares unboxed values
    System.out.println(a <= b); // true : compares unboxed values
    System.out.println(a > b);  // false: compares unboxed values
    System.out.println(a >= b); // true : compares unboxed values
    System.out.println(a != b); // true : compares references
  }
}
```

What does `Integer.valueOf(int)` do?

Listing: What does `java.lang.Integer.valueOf(int)` do?

```java
public final class Integer extends Number implements Comparable<Integer> {
// ...
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }

// ...
        private static class IntegerCache {
            static final int low = -128;
            static final int high;
        static final Integer cache[];

 static {
            // high value may be configured by property
            int h = 127;
// ...
            high = h;
// ...
        }
        }
}
```

- It caches $-128 \ldots 127$, so in this range, it will return the same objects, otherwise its return values are different objects

Comparisons on Boxed Integers within Cache Limits

Listing: Comparisons on Boxed Integers within Cache Limits

```java
/** The same primitive value not necessarily wraps to the same object */
public class IntArghhOh {
  /** The main routine
   *  @param args  we ignore it */
  public static void main(String[] args) {
    Integer a, b;

    a = 1000;
    b = 1000;
    System.out.println(a == b); // false: compares references
    System.out.println(a < b);  // false: compares unboxed values
    System.out.println(a <= b); // true : compares unboxed values
    System.out.println(a > b);  // false: compares unboxed values
    System.out.println(a >= b); // true : compares unboxed values
    System.out.println(a != b); // true : compares references
  }
}
```

What does `Double.valueOf(double)` do?

Listing: What does `java.lang.Double.valueOf(double)` do?

```
public final class Double extends Number implements
    Comparable<Double> {
// ...
    public static Double valueOf(double d) {
        return new Double(d);
    }
// ...
}
```

- It never caches anything, *all* of its return values are different objects

Summary

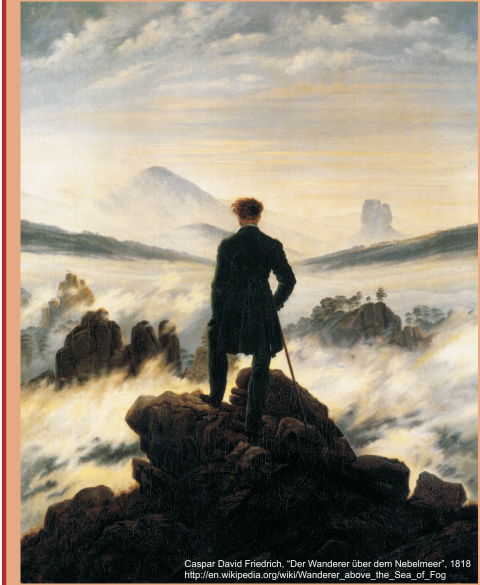
- We have learned about autoboxing namely the automatic wrapping of primitive types into the wrapper classes when objects are required
- We have learned about auto-unboxing namely the automatic unwrapping of wrapper objects into primitive types when the corresponding primitive types are required
- This allows us to put primitive types into lists and maps and treat them as objects
- But it may also have some unintended and un-obvious consequences
- Nowadays, many programmers very liberally use autoboxing
- I say: No! Away with it! If you need an object, wrap it manually by manually calling `<WrapperType>.valueOf(...)` by yourself
- Be aware what's going on in your code.

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China



Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog