Theory: Boxing and unboxing

16 minutes 5 / 10 problems solved

Start practicing

4493 users solved this topic. Latest completion was 3 minutes ago.

§1. Wrapper classes

Each primitive type has a class dedicated to it. These classes are known as **wrappers** and they are **immutable** (just like strings). Wrapper classes can be used in different situations:

- when a variable can be `null` (absence of a value);
- when you need to store values in generic collections (will be considered in the next topics);
- when you want to use special methods of these classes.

The following table lists all primitive types and the corresponding wrapper classes.

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

The table with primitive types and the corresponding wrappers

As you can see, Java provides eight wrapper classes: one for each primitive type. The third column shows the type of argument you need so that you can create an object of the corresponding wrapper class.

§2. Boxing and unboxing

Boxing is the conversion of primitive types to objects of corresponding wrapper classes. **Unboxing** is the reverse process. The following code illustrates both processes:

```
int primitive = 100;
Integer reference = Integer.valueOf(primitive); // boxing
int anotherPrimitive = reference.intValue();    // unboxing
```

Autoboxing and **auto-unboxing** are automatic conversions performed by the Java compiler.

```
double primitiveDouble = 10.8;
Double wrapperDouble = primitiveDouble; // autoboxing
double anotherPrimitiveDouble = wrapperDouble; // auto-unboxing
```

You can mix both automatic and manual boxing/unboxing processes in your programs.

Autoboxing works only when the left and the right parts of an assignment have the same type. In other cases, you will get a compilation error.

Current topic:

✓ Boxing and unboxing

Topic depends on:

✓ Immutability

✓ Objects Stage 3

✓ NPE

Topic is required for:

Object interning

Generic programming

Table of contents:

```
Long n1 = 10L; // OK, assigning long to Long
Integer n2 = 10; // OK, assigning int to Integer

Long n3 = 10; // WRONG, assigning int to Long
Integer n4 = 10L; // WRONG assigning long to Integer
```

§3. Constructing wrappers based on other types

The wrapper classes have constructors for creating objects from other types. For instance, an object of a wrapper class can be created from a string (except for the `Character` class).

```
Integer number = new Integer("10012"); // an Integer from "10012"
Float f = new Float("0.01");           // a Float from "0.01"
Long longNumber = new Long("100000000"); // a Long from "100000000"
Boolean boolVal = new Boolean("true");   // a Boolean from "true"
```

You can also create wrapper objects using special methods:

```
Long longVal = Long.parseLong("1000");      // a Long from "1000"
Long anotherLongVal = Long.valueOf("2000"); // a Long from "2000"
```

If the input string has an invalid argument (for instance, `"1d0o3"`), both of these methods throw the `NumberFormatException`.

Note, since Java 9, the constructors are deprecated. You should use special methods to create objects of the wrapper classes.

§4. Comparing wrappers

Just like for any reference type, the operator `==` checks whether two wrapper objects are actually equal, i.e. if they refer to the same object in memory. The method `equals` , on the other hand, checks whether two wrapper objects are meaningfully equal, for example, it checks if two wrappers or strings have the same value.

```
Long i1 = Long.valueOf("2000");
Long i2 = Long.valueOf("2000");
System.out.println(i1 == i2);      // false
System.out.println(i1.equals(i2)); // true
```

Do not forget about this feature when working with wrappers. Even though they correspond to the primitive types, wrapper objects are reference types!

§5. NPE when unboxing

There is one possible problem when unpacking. If the wrapper object is `null`, the unpacking throws `NullPointerException`.

```
Long longVal = null;
long primitiveLong = longVal; // It throws NPE
```

To fix it, we can add a conditional statement that produces a default value:

```
long unboxed = val != null ? val : 0; // No NPE here
```

This code does not throw an exception.

Another example is arithmetic operations on `Integer`, `Long`, `Double` and other numeric wrapper types. They may cause an NPE since there is auto-unboxing involved.

```
Integer n1 = 50;
Integer n2 = null;
Integer result = n1 / n2; // It throws NPE
```

§6. Primitive types vs wrapper classes

In this topic, we've taken a look at wrapper classes for primitive data types. Wrapper classes allow us to represent primitive types as objects, that are reference data types.

Here are some important points to keep in mind:

- processing values of primitive types is faster than processing wrapper objects;
- wrappers can be used when you need `null` as a no-value indicator;
- primitive types cannot be used in standard collections (like lists, sets, or others), but wrappers can.

Report a typo

373 users liked this theory. 7 didn't like it. What about you?

Start practicing

Comments (15) Hints (0) Useful links (0) Show discussion