Programming in Java – Primitive type wrappers, Enumerations, and Generics

Paolo Vercesi
Technical Program Manager

Agenda

Primitive type wrappers

Enumerations

Generics

Primitive type wrappers

Primitive types

- `byte`
- `short`
- `int`
- `long`
- `character`
- `float`
- `double`
- `boolean`
- `void`

✓ Primitive types are not objects!

✓ Primitive types are used for performance reasons, however many situations require an object

✓ Most Java classes have methods that works on Objects and not on primitive type.

Primitive types are not objects

```
public interface List {

    boolean isEmpty();

    int getSize();

    boolean contains(Object value);

    Object[] getValues();

    Object get(int index);

    void add(Object value);

    void insertAt(int index, Object value);

    void remove(int index);

    int indexOf(Object value);
}
```

- ✓ Can I use primitive types with classes implementing this List interface?
- ✓ The same interface and consequently the same implementation cannot be used for primitive types

A List interface for the int type

```java
public interface IntList {

    boolean isEmpty();

    int getSize();

    boolean contains(int value);

    int[] getValues();

    int get(int index);

    void add(int value);

    void insertAt(int index, int value);

    void remove(int index);

    int indexOf(int value);
}
```

- ✓ We would need one interface for the int type, one for the long type, one for the short type, and so forth
- ✓ In some cases, Java uses this approach. Why?
- ✓ Only for performance reasons. More on this when we will talk about Streams.

Primitive type wrappers

- `byte`
- `short`
- `int`
- `long`
- `character`
- `float`
- `double`
- `boolean`

- `Byte`
- `Short`
- `Integer`
- `Long`
- `Character`
- `Float`
- `Double`
- `Boolean`

✓ There exist one wrapper class for each primitive type

✓ Wrappers are classes that wrap primitive types within an object

From primitive value to wrapper object

The static factory method valueOf() is the recommended way to convert a primitive value to an object

```
int i = 60;
Integer i1 = Integer.valueOf(i);
Integer i2 = new Integer(i);
```

The use of primitive type wrapper's constructors is deprecated.

Why?

Hint: consider the boolean case

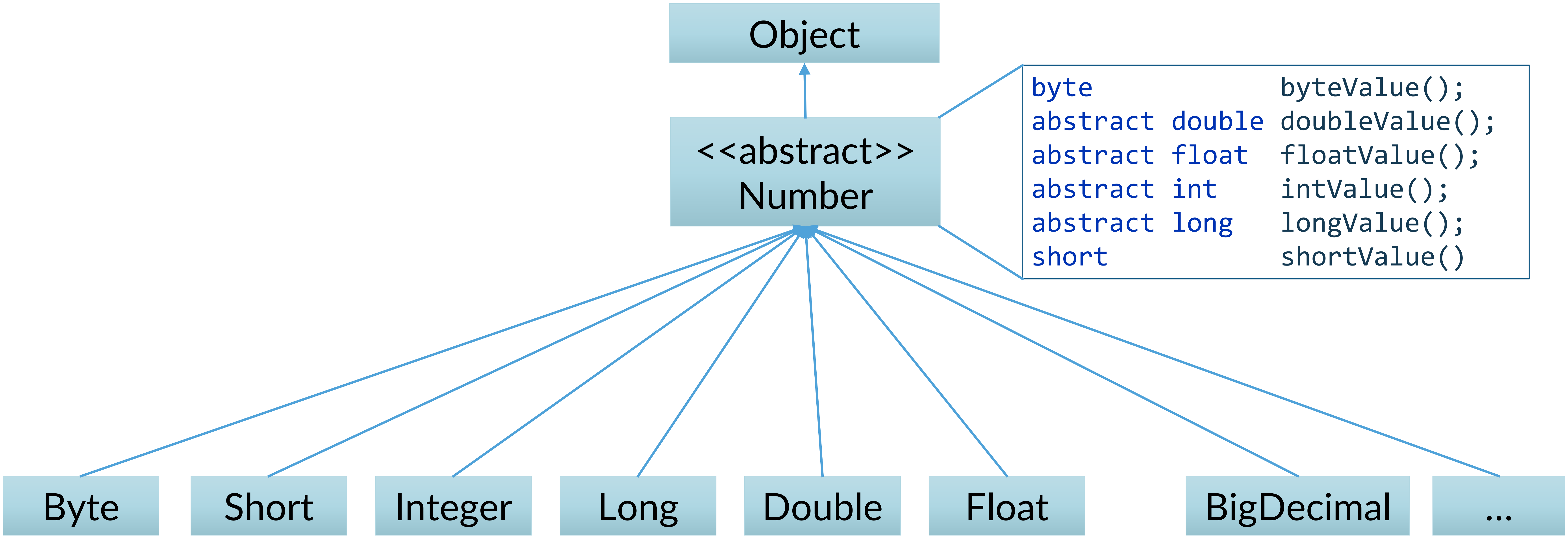
```
boolean b = true;
Boolean b1 = Boolean.valueOf(b);
Boolean b2 = new Boolean(b);
```

From wrapper object to primitive value

```
Boolean b = Boolean.FALSE;
boolean b1 = b.booleanValue();

Character c = Character.valueOf('a');
char c1 = c.charValue();
```

What about numerical values?

The Number hierarchy



Boxing and unboxing

Also known as auto-boxing and auto-unboxing

```
Boolean b = false;
boolean b1 = b;

Character c = 'a';
char c1 = c;

int i = 60;
Integer i1 = i;
Double d1 = i;
Double d1 = i1.doubleValue();
int i2 = i + i1;
```

Boxing is the process by which a primitive type is automatically wrapped into its equivalent type wrapper whenever an object of that type is needed.

There is no need to explicitly construct such an object.

Unboxing is the process by which the value of a boxed object is automatically extracted from a type wrapper when its value is needed.

There is no need to call a method such as `intValue()` or `doubleValue()`.

Unboxing can lead to NullPointerExceptions

Caching of wrapper objects

If the value *p* being boxed is the result of evaluating a constant expression (§15.29) of type `boolean`, `byte`, `char`, `short`, `int`, or `long`, and the result is `true`, `false`, a character in the range `'\u0000'` to `'\u007f'` inclusive, or an integer in the range `-128` to `127` inclusive, then let *a* and *b* be the results of any two boxing conversions of *p*. It is always the case that *a* == *b*.

From The Java Language Specification, Java SE 17 Edition, p. 123

```
public static void main(String[] args) {
    System.out.println(Integer.valueOf(127) == Integer.valueOf(127));
    System.out.println(Integer.valueOf(128) == Integer.valueOf(128));
}
```

The operator == tells you if two references point to the same object. No unboxing is performed here.

Assignment

Explore the API of the primitive type wrappers.
If you haven't yet done so.

Enumerations

Enumerations

```
enum Degree {
    HIGH_SCHOOL, BACHELOR, MASTER, PHD
}
```

Each enumeration constant is a public static final member of the Degree class

An enumeration declaration
1. is a list of named constants
2. that define a new data type
3. and its legal values.

Once it is declared, an enumeration cannot be changed at runtime.

```
Degree d1 = Degree.PHD;
Degree d2 = Degree.BACHELOR;

if (d1 == d2) {
    System.out.println("This seems a bit unusual!");
}
```

You don't instantiate an enumeration, but you can reference its members

Enumerations in switch statements

In switch statements the enumeration constants don't need to be qualified by their enumeration type name

```
Degree d = getDegree();

switch (d) {
    case HIGH_SCHOOL -> System.out.println("High School");
    case BACHELOR -> System.out.println("Bachelor");
    case MASTER -> System.out.println("Master");
    case PHD -> System.out.println("PhD");
}
```

Arrow notation

No need for break statements

Enumerate enumerations

Enumerations automatically get two static methods, one to enumerate the constants and one to get a constant from its name

```
public static enum-type [ ] values( )
public static enum-type valueOf(String str )
```

Each enumeration constant has an ordinal value

```
final int ordinal( )
```

```
Degree d1 = Degree.PHD;
Degree d2 = Degree.valueOf("PHD");

if (d1 == d2) {
    System.out.println("This looks ok!");
}

for (Degree dd : Degree.values()) {
    System.out.println(dd.ordinal(dd) + " " + dd);
}
```

Enumerations are first class classes

```
enum Degree {
    HIGH_SCHOOL(5), BACHELOR(3), MASTER(2), PHD(3);

    private final int duration;

    Degree(int duration) {
        this.duration = duration;
    }

    public int getDuration() {
        return duration;
    }
}
```

They can have fields

They can have constructors

They can have methods

Generics

Generalized classes

```java
public class GeneralizedStack {

     public int getSize();

     public Object top() {…}

     public Object pop() {…}

     public void push(Object value) {…}

}
```

Before the introduction of generics in Java 5, generalized classes, interfaces and methods operated with references to Object instances, with consequent problems of type safety

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

    GeneralizedStack stack = new GeneralizedStack();

    stack.push("Hello,");
    stack.push("World!");
    stack.push(new Object());

    while (stack.getSize() > 0) {
        String text = (String) stack.pop(); Runtime exception
        System.out.println(text);
    }
}
```

We had to rely on inherently unsafe casting

Specialized classes

```java
public class StringStack {

    private final GeneralizedStack data =
            new GeneralizedStack();

    public int getSize() {
        return data.getSize();
    }

    public String top() {
        return (String) data.top();
    }

    public String pop() {
        return (String) data.pop();
    }

    public void push(String value) {
        data.push(value);
    }
}
```

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

    StringStack stack = new StringStack();

    stack.push("Hello,");
    stack.push("World!");
    stack.push(new Object());

    while (stack.getSize() > 0) {
        String text = stack.pop();
        System.out.println(text);
    }
}
```

Compile time error

No need for class casting, but specialized classes required boilerplate code and a considerable use of cast operations.

What are Generics?

```java
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}
}
```

Java 5 introduced the concept of parameterization of interfaces, classes and methods. A parameterized interface or class is called parameterized type or generic.

In generic classes, generic interfaces, and generic methods the type of data upon which they operate is specified as a parameter

Parameterized types are used to improve type safety when compared with the use of Object and they are used to reduce boilerplate code.

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

    Stack<String> stringStack = new Stack<>();

    stringStack.push("Hello,");
    stringStack.push("World!");
    stringStack.push(new Object());

    while (stringStack.getSize() > 0) {
        String text = stringStack.pop();
        System.out.println(text);
    }
}
```

Compile time error

Parameters bounding

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}
}
```

The parameter T can be replaced by any class type.

```
public class NumberStack<N extends Number> extends Stack<N> {

    double average() {
        double sum = 0;
        for (Number number : data) {
            sum += number.doubleValue();
        }
        return sum / getSize();
    }
}
```

N is a bounded parameter, N can be replaced only by the superclass Number or by a subclass of the superclass Number.

Multiple bounding & intersection types

The parameter T can be bounded to a class and to any number of interfaces

In that case, the bounding class must be specified first and T will be bounded to the intersection type

```
public class Stack<T extends KeylessCar & AutonomousCar> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}
}
```

Wildcard arguments 1/3

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}

    public void sameSize(Stack<T> other) {
        return getSize() == other.getSize();
    }
}
```

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

    Stack<String> stringStack = new Stack<>();
    Stack<Double> doubleStack = new Stack<>();

    doubleStack.sameSize(stringStack);
}
```

This code doesn't compile

The `sameSize` method expects `Stack<Double>`

Wildcard arguments 2/3

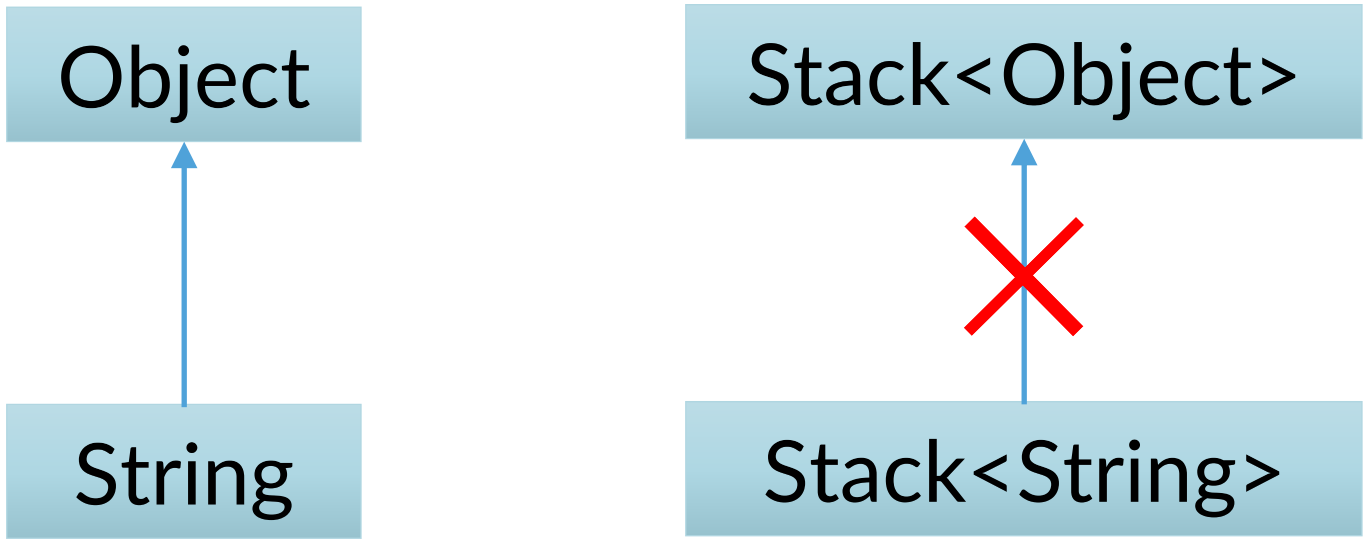
```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}

    public void sameSize(Stack<Object> other) {
        return getSize() == other.getSize();
    }
}
```

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

    Stack<String> stringStack = new Stack<>();
    Stack<Double> doubleStack = new Stack<>();

    doubleStack.sameSize(stringStack);
}
```

The main method still doesn’t compile

String is a subclass of Object

Stack<String> is not a “subclass” of Stack<Object>



Wildcard arguments 3/3

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}

    public void sameSize(Stack<?> other) {
        return getSize() == other.getSize();
    }
}
```

The `sameSize` method now expects a `Stack<?>` that means a Stack with any parameterization

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

    Stack<String> stringStack = new Stack<>();
    Stack<Double> doubleStack = new Stack<>();

    doubleStack.sameSize(stringStack);
}
```

This code compiles

Multiple parameters

A generic can define multiple type parameters

```
public class Pair<F, S> {

    private final F first;
    private final S second;

    public Pair(F first, S second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public String toString() {
        return "Pair{first=" + first + ", second=" + second + '}';
    }
}
```

Generic methods 1/3

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}
}
```

Methods inside a generic class can make use of a class type parameter and are, therefore, automatically generic relative to the type parameter

Generic methods 2/3

```
public class Stack<T> {

    public int getSize() {…}

    public T top() {…}

    public T pop() {…}

    public void push(T value) {…}

    public <O extends T> void pushAll(Stack<O> other) {
        while (other.getSize() > 0) {
            push(other.pop());
        }
    }
}
```

Type parameters are declared before the return type

Type parameters are used in the argument list

Generic methods 3/3

```
public class Stack<T> {

    public int getSize() {…}

    public T top() {…}

    public T pop() {…}

    public void push(T value) {…}

    public void pushAll(Stack<? extends T> other) {
        while (other.getSize() > 0) {
            push(other.pop());
        }
    }
}
```

This class is equivalent to the previous one

Wildcards are preferred, they make the code more concise

Generic interfaces

```
public interface Stack<T> {

    int getSize();

    void push(T value);

    T top();

    T pop();
}
```

A generic interface is declared in the same way as a generic class

Local variable type inference

```
Stack<String> stringStack1 = new Stack<>();
var stringStack2  = new Stack<String>();
```

The second version is shorter and it should be preferred

Assignment

Quasi-trivial assignment

```
public interface Collection {

    boolean isEmpty();

    int getSize();

    boolean contains(Object object);

    Object[] getValues();
}

public interface Stack extends Collection {

    void push(Object object);

    Object pop();

    Object top();
}
```

```
public interface List extends Collection {

    void add(Object object);

    Object get(int index);

    void insertAt(int index, Object object);

    void remove(int index);

    int indexOf(Object object);

}
```

Rewrite the Collection, Stack and List interfaces, so that they become generic. Update the implementation.

Hint: `getValues()` is not trivial. Take a look at `java.lang.reflect.Array`

Thank you!

esteco.com