Programming in Java – Part 05 – Enumerations, record, primitive type wrappers, and generics

Paolo Vercesi

ESTECO SpA

Agenda

- Enumerations
- Records
- Primitive type wrappers
- Generics

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, i.e., you cannot add or remove values

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

if (d1 == d2) {
    IO.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 -> IO.println("High School");
    case BACHELOR -> IO.println("Bachelor");
    case MASTER -> IO.println("Master");
    case PHD -> IO.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) {
    IO.println("This looks ok!");
}

for (Degree dd : Degree.values()) {
    IO.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

Digression – Constants before Enum

```
public class GridBagConstraints implements Cloneable, java.io.Serializable {

    /** Specifies that this component is the next-to-last component in its …*/
    public static final int RELATIVE = -1;

    /** Specifies that this component is the
     * last component in its column or row. */
    public static final int REMAINDER = 0;

    /** Do not resize the component. */
    public static final int NONE = 0;

    /** Resize the component both horizontally and vertically. */
    public static final int BOTH = 1;

    /** Resize the component horizontally but not vertically. */
    public static final int HORIZONTAL = 2;

    /** Resize the component vertically but not horizontally. */
    public static final int VERTICAL = 3;

    /** Put the component in the center of its display area. */
    public static final int CENTER = 10;

    /** Put the component at the top of its display area,
     * centered horizontally. */
    public static final int NORTH = 11;

    /** Put the component at the top-right corner of its display area. */
    public static final int NORTHEAST = 12;
```

Records

Record

The record keyword defines classes designed to be data transfer objects also called data carriers

```
public record Dish(String name, boolean vegetarian, int calories, Type type) {

    public enum Type {MEAT, FISH, OTHER}

    @Override
    public String toString() {
        return name;
    }
}
```

Record classes automatically generate

- *Immutable fields*
- *A canonical constructor*
- *An accessor method for each element*
- *The equals() method*
- *The hashCode() method*
- *The toString() method*

When you cannot use records

1. *When you have a mutable state to implement some internal logic*
    - *records are data carriers*
2. *In general, when you need mutability*
    - *all fields are final by default*
3. *When you need inheritance hierarchy*
    - *records are final classes*
4. *When you need compatibility with older frameworks*
    - *they could require an empty constructor*

Reference

A good compact reference

https://blogs.oracle.com/javamagazine/post/java-records-constructor-methods-inheritance

Primitive type wrappers

Primitive types

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

✓ *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

```
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);
```

Additional content - 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) {
    IO.println(Integer.valueOf(127) == Integer.valueOf(127));
    IO.println(Integer.valueOf(128) == Integer.valueOf(128));
}
```

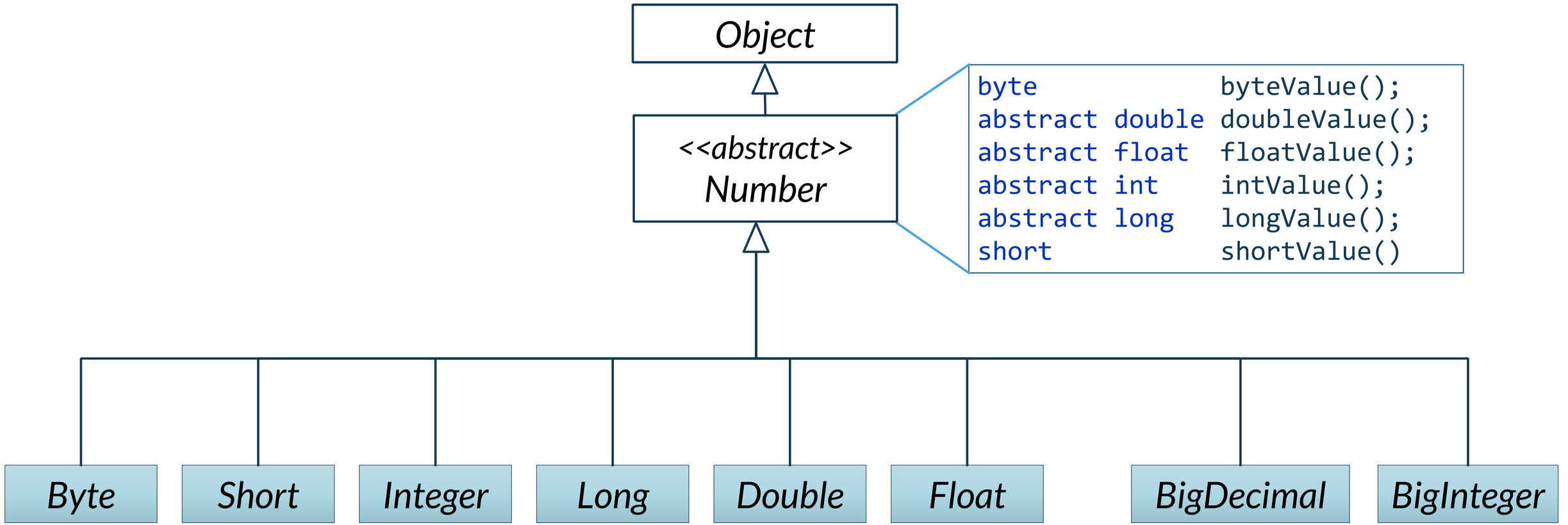
The operator == tells you if two references point to the same object.
No unboxing is performed here.

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

Generics

Generalized classes

```
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

```
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
        IO.println(text);
    }
}
```

We had to rely on inherently unsafe casting

Specialized classes

```
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);
    }
}
```

```
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();
        IO.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?

```
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

```
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();
        IO.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.

Wildcard arguments 1/3

```
public class Stack<T> {

  public int getSize() {…}

  public void push(T value) {…}

  public T top() {…}

  public T pop() {…}

  public boolean 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>`

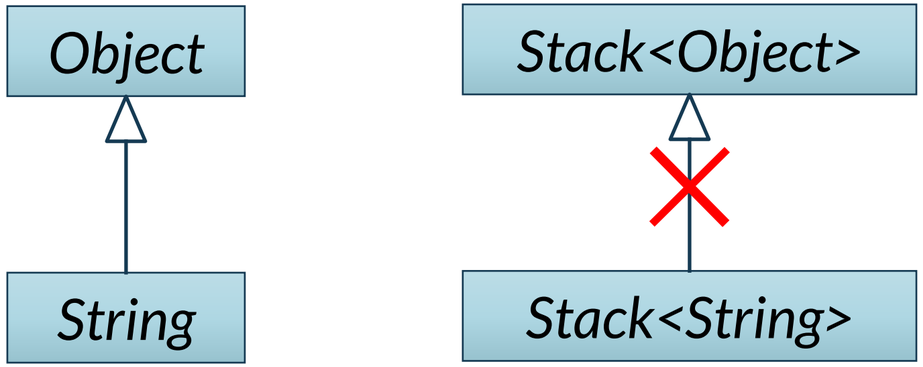
Wildcard arguments 2/3

```
public class Stack<T> {

  public int getSize() {…}

  public void push(T value) {…}

  public T top() {…}

  public T pop() {…}

  public boolean 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, but
Stack<String> is not a "subclass" of Stack<Object>



Suppose Stack<String> were a "subclass" of Stack<Object>

```
Stack<String> ss = new Stack<>();
Stack<Object> so = ss;
so.add(new Object());
String x = ss.get(0);
```

Wildcard arguments 3/3

```
public class Stack<T> {

    public int getSize() {…}

    public void push(T value) {…}

    public T top() {…}

    public T pop() {…}

    public boolean 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

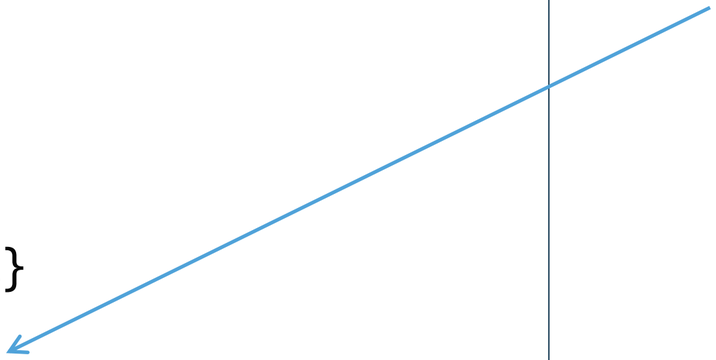
Type parameters are used in the argument list

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 could be preferred

Self assessment

Quiz 1: Enumerations

1. **What does an enum in Java represent?**
   A. *A primitive type*
   B. *A list of integers*
   C. *A class with a fixed set of named constants*
   D. *A method returning constants*

2. **Which of these is true about enumeration constants?**
   A. *They are instantiated with new*
   B. *They are primitive literals*
   C. *They are public static final instances of the enum type*
   D. *They can be reassigned at runtime*

3. **In an enum declaration:**
   `enum Degree { HIGH_SCHOOL, BACHELOR, MASTER, PHD }`
   What is the ordinal of MASTER?
   A. *0*
   B. *1*
   C. *2*
   D. *3*

Quiz 1: Enumerations

4. **What is the result of this code?**

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

   A. *true*
   B. *false*
   C. *Compilation error*
   D. *Depends on JVM*

5. **Which of these is a valid feature of enums in Java?**
   A. *They can have fields, constructors, and methods*
   B. *They can be subclassed*
   C. *They must be abstract*
   D. *They can only hold primitive constants*

Quiz 2: Records

1. **What is the primary purpose of a Java record?**
   A. *To create mutable data containers*
   B. *To define data carrier classes with automatically generated methods*
   C. *To allow inheritance from abstract classes*
   D. *To replace traditional interfaces*
2. **Which methods are automatically generated for a record?**
   A. *equals(), hashCode(), toString(), accessors, and a canonical constructor*
   B. *Only toString() and a no-arg constructor*
   C. *All methods from Object*
   D. *None; you must define them manually*

Quiz 2: Records

3. **Given:**
   ```
   public record Dish(String name, boolean vegetarian, int calories) {}
   ```
   What is true about its fields?
   A. *They are mutable by default*
   B. *They are public variables*
   C. *They are protected*
   D. *They are private final and accessible via generated accessors*

4. **Why might you override a method (like toString()) in a record?**
   A. *To break immutability*
   B. *To prevent accessor generation*
   C. *To disable auto-generated constructors*
   D. *To customize how the record is represented or logged*

Quiz 3: Primitive type wrappers

1. **What is the main purpose of wrapper classes in Java?**
   A. *To improve performance of primitives*
   B. *To allow primitives to be used where objects are required*
   C. *To replace primitives completely*
   D. *To avoid autoboxing*

2. **Which pair is not a valid primitive-to-wrapper match?**
   A. `int` → *Integer*
   B. *char* → *Character*
   C. *boolean* → *Boolean*
   D. *double* → *Decimal*

3. **What is autoboxing?**
   A. *Automatically converting a wrapper object to its primitive*
   B. *Automatically converting a primitive value to its wrapper object*
   C. *Manually wrapping values using valueOf()*
   D. *Converting between unrelated numeric types*

Quiz 3: Primitive Type Wrappers

4. **Why is `Integer.valueOf()` preferred over `new Integer()`?**
   A. *It uses object caching and avoids unnecessary object creation*
   B. *It's shorter syntax*
   C. *It returns a primitive*
   D. *It performs unboxing automatically*

5. **What's the output of:**

```
IO.println(Integer.valueOf(127) == Integer.valueOf(127));
IO.println(Integer.valueOf(128) == Integer.valueOf(128));
```

   A. *true true*
   B. *false false*
   C. *true false*
   D. *false true*

Quiz 4: Generics

1. **What problem were generics designed to solve in Java?**
   A. *Lack of polymorphism*
   B. *Lack of inheritance*
   C. *Type safety issues caused by Object references*
   D. *Performance of collections*

2. **Given:**
   ```
   Stack<String> stack = new Stack<>();
   stack.push("Hi");
   stack.push(new Object());
   ```
   What happens?
   A. *Compiles and runs fine*
   B. *Compile-time error due to type mismatch*
   C. *Runtime ClassCastException*
   D. *Autoboxing converts the Object*

Quiz 4: Generics

3. **What does the syntax `<T extends Number>` mean?**
   A. *T must be Number*
   B. *T can be any class*
   C. *T can be Number or any subclass of it*
   D. *T can be any type implementing Comparable*

4. **Why does this method not compile?**
   `public boolean sameSize(Stack<Object> other)`
   when comparing `Stack<String>` and `Stack<Double>`?
   A. *Java generics are "invariant": Stack<String> is not a subclass of Stack<Object>*
   B. *Stack doesn't support comparison*
   C. *The compiler doesn't allow wildcards*
   D. *Stack is abstract*

Quiz 4: Generics

1. **What does the wildcard `?` in `Stack<?>` mean?**
   A. *Stack of Objects only*
   B. *Stack with no elements*
   C. *Stack of unknown or any type parameterization*
   D. *Stack restricted to numeric types*

Correct answers

Quiz 1

1. *C*
2. *C*
3. *C*
4. *A*
5. *A*

Quiz 2

1. *B*
2. *A*
3. *D*
4. *D*

Quiz 3

1. *B*
2. *D*
3. *B*
4. *A*
5. *C*

Quiz 4

1. *C*
2. *B*
3. *A*
4. *A*
5. *C*

Thank you!

esteco.com