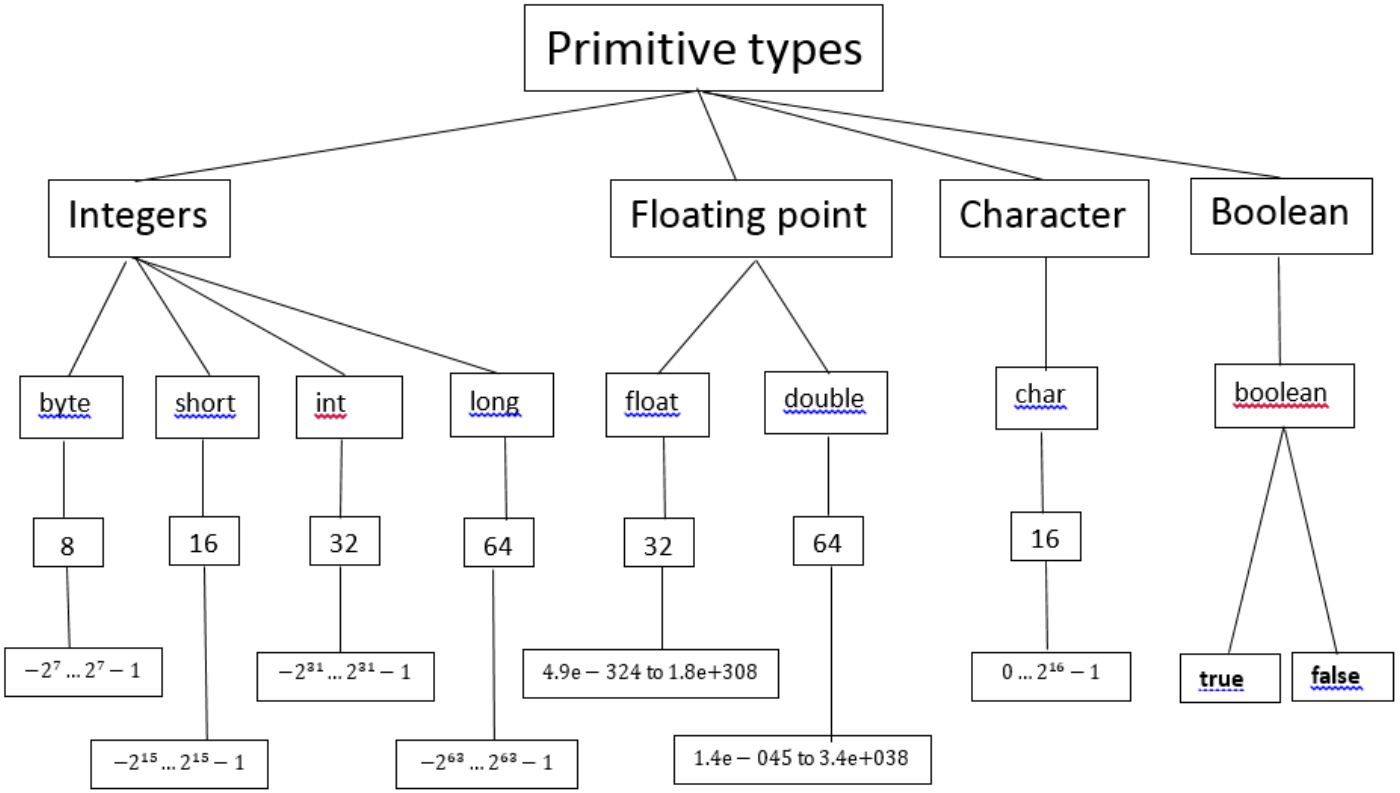
Data types and operators

GEORGIA'S INNOVATION
AND TECHNOLOGY AGENCY

skillUp
The expert of IT education

Primitive data types

The values of the primitive data types are placed in the stack. These types provide greater performance compared to the objects. For each primitive type there are wrapper classes that encapsulate data primitive types to objects that reside in the heap. 8 primitive data types are defined.



Casting of numeric types

Casting of the primitive types is executed automatically in arithmetic expressions:

byte → short → int → long → float → double

If the operands ***a*** and ***b*** are combined by binary operator, before it is executed, both operands are converted to the same type of data in the following way:

if one of the operators is of type *double*, the second also is converted to *double*;

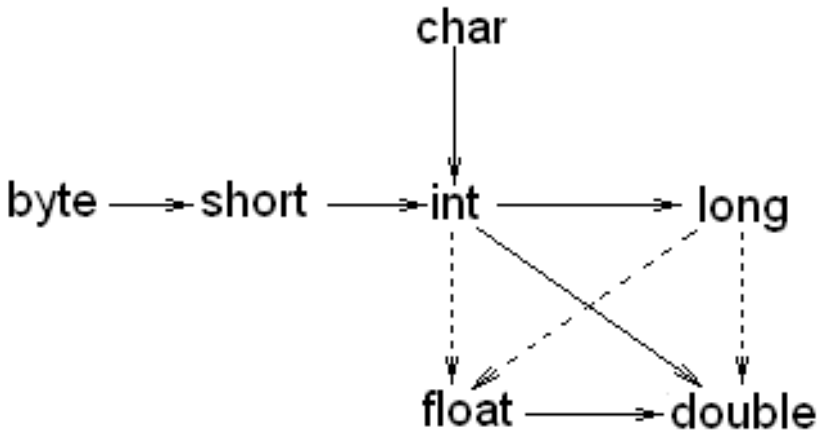
if one of the operators is of type *float*, the second one also is converted to *float*;

if one of the operators is of type *long*, the second one also is converted to *long*;

if one of the operators is of type *int*, the second one also is converted to *int*;

Casting of numeric types

Permitted type conversions are shown in the figure:



The solid lines show the transformation performed without loss of information. This casting is performed implicitly. Transformations when loss of information may occur, are called *casting*. They are shown by dashed lines.

Casting of numeric types

For a narrowing casting it is necessary to make explicit transformation of the form: ***(type) value***. For example:

```
byte b = (byte)128; //cast int to byte
```

Casting of numeric types

```
public class TypeByte {
        public static void main(String[] args) {
                byte b = 1, b1 = 1 + 2;
                final byte B = 1 + 2;
//b = b1 + 1; //error cast
/* b1 – variable at the time of execution of code b = b1 +
1; may change, and the expression of b1 + 1 may exceed the
allowable size of byte- type */
                b = (byte)(b1 + 1);
                b = B + 1; // works
/* B - constant, its value is determined, the compiler
calculates the value of the expression B + 1, and if it
does not exceed the size of byte type, the error does not
occur */
                final int I = 3;
                b = I;  // works
/*I – constant. The compiler checks to see whether it
exceeds the values-of allowable size for the type of byte,
if not exceed, the error does not occur */
                final int I2 = 129;
//b=I2; // Error type casting, as 129 is greater than 127
                b = (byte) I2;
        }
}
```

Variable

Variable names can not begin with a number, names can not be used symbols of arithmetic and logical operators, as well as the character '#'.

Applying symbols '$' or '_' acceptable, including a first position and name.

The variable of primitive type declared as a member of the class, keeps zero.

If the variable is declared as a local variable in the method, before using it must necessarily be initialized because it is not initialized by default to zero.

The scope and lifetime of a variable is limited to the block {}, in which it is declared.

Operators

Java operators practically coincide with the operators of C++, and have the same priority.

Since pointers in Java does not support, then there are no operators *, &, ->, *delete* to work with them.

Operators work with the primitive types of objects and wrapper classes on the primitive types.

The operators + and + = also produce action concatenation with operands of type *String*.

Operators

Java have several types of operator: simple assignment, arithmetic, unary, equality and relational, conditional, type comparison, bitwise and bit shift.

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Operators

Let's look to ***ternary*** operator (shorthand for if-then-else statement) and ***instanceof*** operator that compares an object to a specified type. Logical operations are performed only on the values of boolean and Boolean types (true or false).

```
booleanvalue ? first : second
```

If ***booleanvalue*** is **true**, the expression is evaluated to first, otherwise to second.

Operators

The instanceof operator returns true, if the object is instances of the given class. For example, for the inheritance hierarchy:

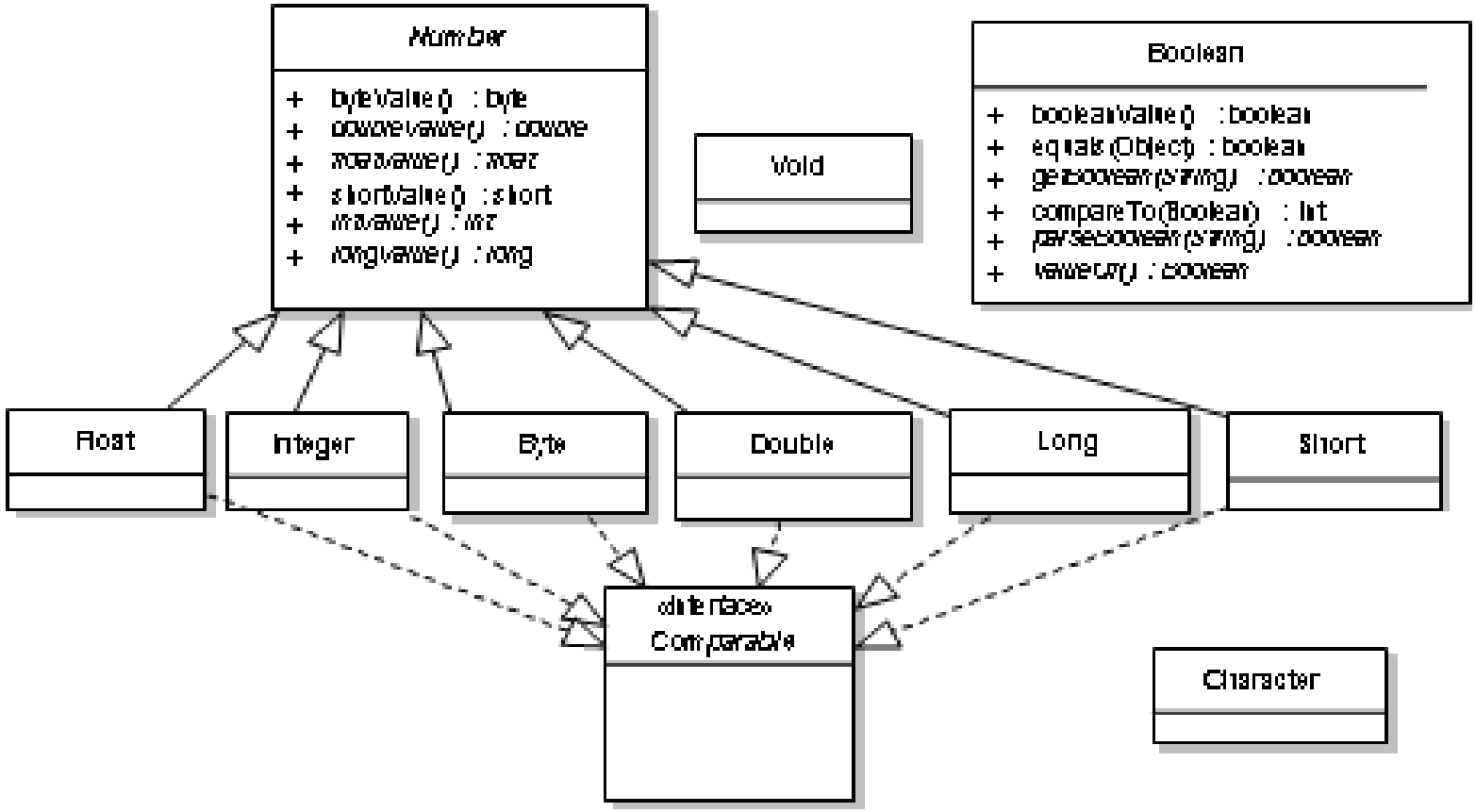
```
class A extends Object {}
class B extends A {}
class D extends B {}
class C extends a {}

void methodName(A obj) {
if (obj instanceof B) {/*implementation for B and D*/
} else if (obj instanceof C){/*implementation for C
*/
} else {/*implementation for A*/}
}
```

The result of the ***instanceof*** operator is true if the object is the instance of a given type or one of its subclasses, but not conversely-mouth. Using operator ***instanceof*** for Object class will always true, because every class is inherited from the class Object. The result of applying this operator to a reference to a ***null*** value is always ***false***, because ***null*** can not be assigned to any type.

Wrapper classes

Apart from primitive types in the Java widely used corresponding wrapper classes from the package java.lang: ***Boolean, Character, Integer, Byte, Short, Long, Float, Double***. Objects of these classes may store the same values as the corresponding primitive types.



Wrapper classes

Variable of primitive type can be converted to the corresponding object by passing it to the constructor when declaring the instance. Instance of class can be converted to any of primitive type by calling methods **value().**

```
package chapter02;
public class CastTypes {
 public static void main(String[] args) {
  Float f1 = new Float(10.71); // double to Float
  String s1 = Float.toString(0f); // float to String
  String s2 = String.valueOf(f1); // Float to String
  Byte b = Byte.valueOf("120"); // String to Byte
  double d = b.doubleValue(); // Byte to double
  byte b0=(byte)(float)f1;// Float to byte
 }
}
```

Autoboxing

Java has the automatic data encapsulation of primitive types to the appropriate wrapper objects and back (autoboxing). There is no need to use operator ***new***. For Example:

Integer field = 7;

Unboxing – is the process of extracting from an wrapper-object value of the primitive type. This object can be used in arithmetic operations, but packing / unpacking is a resource-intensive process.

```
// sample # 4 : autoboxing & unboxing: NewProperties.java
package chapter02;
public class AutoboxingClass {
  public static void main(String[] args) {
     Integer a = 7; //creating object+autoboxing
     Integer b = ++a; //unboxing+action+autoboxing
     int c = 3;
     b = a + b + c;
  }
}
```

What is the result?

```
int i = 128;
Integer oa = i;
Integer ob = i;

System.out.println("oa==i " + (oa == i));
System.out.println("ob==i " + (ob == i));
System.out.println("oa==ob " + (oa == ob));
System.out.println("equals ->" + oa.equals(i)
+ ob.equals(i)
+ oa.equals(ob));
```

Autoboxing

```
int i = 128;//change to 127  !!!
Integer oa = i; //creating object+autoboxing
Integer ob = i;
System.out.println("oa==i " + (oa == i));// true
System.out.println("ob==i " + (ob == i));// true
System.out.println("oa==ob " + (oa == ob));// false(different links)
System.out.println("equals ->" + oa.equals(i)
+ ob.equals(i)
+ oa.equals(ob));// true
```

equals() method compares values of the objects, but not the object references. Therefore, the challenge **oa.equals (ob)** returns **true**.

Autoboxing

The value of the primitive type can be passed to the method **equals()**. However, the reference to the primitive type can not call methods:

```
boolean b = i.equals(oa);// compile error
```

Number

It is possible to create objects and arrays, preserving the different primitive types without mutual casting, by reference to the class **Number**:

```
Number n1 = 1;
Number n2 = 7.1;
Number array[] = {71, 7.1, 7L};
```

Important rule

The variable of primitive type is always passed to the method by value, and the variable of wrapper class by the link.

Question?