IT602: Object-Oriented Programming

Lecture - 04

Operators & Expressions

Arpit Rana

27th Jan 2022

Conversions

A ***`type`*** conversion can be applied to values -

- Some must be ***explicitly*** stated in the program, while others are performed ***implicitly***.
- Some can be checked at ***compile time*** to guarantee their validity at runtime, while others may require an extra check at ***runtime***.

Widening and Narrowing Primitive Conversions

`type` conversions are categorised as -

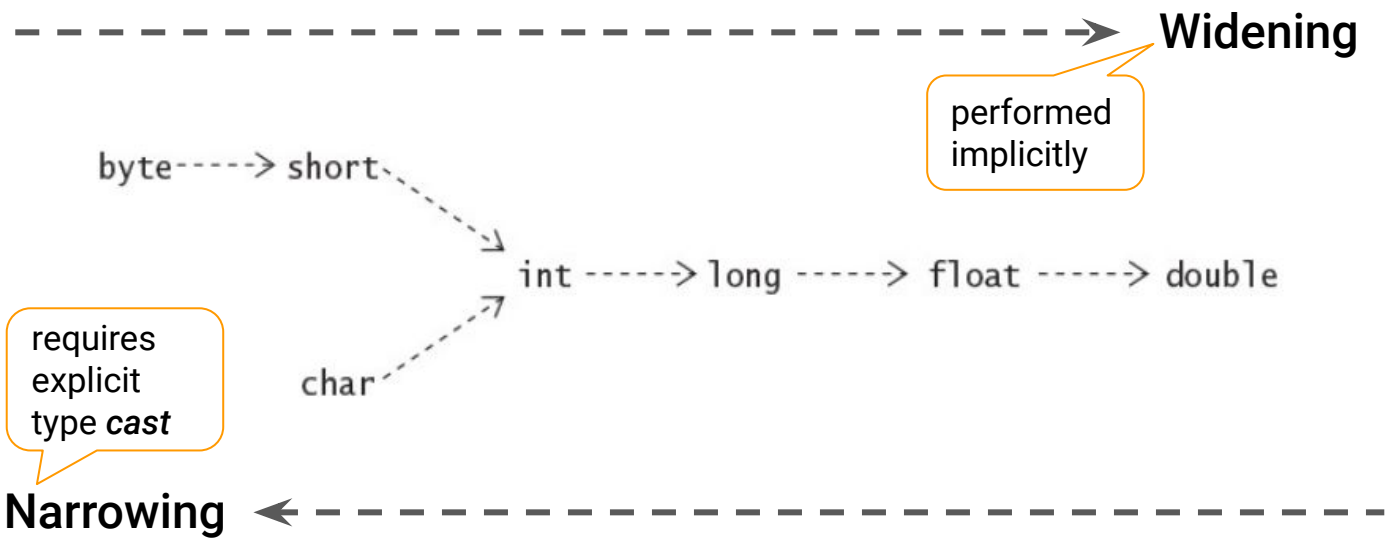
- ***Widening*** (*for primitive data types*): converting a value of narrower data type to a wider data type,

  e.g. *int —> long, float —> double, int —> float*.

- ***Narrowing*** (*for primitive data types*): converting from a wider data type to a to a narrower primitive type.

  e.g. *float —> long, float —> short*

Widening and Narrowing Primitive Conversions



- All conversions between **char** and, **byte** and **short** are considered as narrowing (unsigned char to signed byte or short may result in LoI).

Widening and Narrowing Reference Conversions

The ***`subtype-supertype`*** relationship between *reference types* determine which conversions are permissible between them.

- ***Widening*** (*for reference types*): converting up a type hierarchy, i.e. from subtype to supertype (also called ***upcasting***).

```
Object obj = “Upcast me”;
```

- ***Narrowing*** (*for reference types*): converting down a type hierarchy, i.e. from supertype to subtype (also called ***downcasting***).

```
String str = (String) obj;
```

Widening and Narrowing Reference Conversions

The ***`subtype-supertype`*** relationship between *reference types* determine which conversions are permissible between them.

- Compiler may reject casts that are illegal or may sometimes issue an unchecked warning.
- ***Narrowing*** requires a runtime check and can throw a `ClassCastException` if the conversion is not legal.

Boxing and Unboxing Conversions

It allows interoperability between primitive values and their representation as objects of the wrapper types.

- ***Boxing***: converting the value of a primitive type to a corresponding value of its wrapper type.

```
Integer iRef = 10;          // Boxing: Integer <--- int
System.out.println(iRef.intValue() == 10)     // true
```

- ***Unboxing***: converting the value of a wrapper type to a value of its corresponding primitive type.

```
int i = iRef;               // Unboxing: int <--- Integer
System.out.println(iRef.intValue() == i)      // true
```

- Unboxing a wrapper reference that has a `null` value results in a `NullPointerException`.

Type Conversion Contexts

There are mainly four conversion contexts -

- **Assignment conversions**: an expression (or its value) is assignable to the target variable if the type of the expression can be converted to the type of the target variable by an assignment conversion.

```
byte b = 10;        // Narrowing conversion: byte <--- int

byte b = 324;       // CTE
```

Type Conversion Contexts

There are mainly four conversion contexts -

- **Method Invocation conversions**: involves converting each argument value in a method or constructor call to the type of the corresponding formal parameter in the method or constructor declaration.

```
Character space1 = 32;
// Assignment: (1) implicit narrowing followed by (2) boxing


Character space2 = Character.valueOf(32);
//  CTE: call signature - valueOf(char)


Character space2 = Character.valueOf((char)32); //OK
Character space3 = Character.valueOf(space1);
// Ok, Unboxing
```

Type Conversion Contexts

There are mainly four conversion contexts -

- **Casting conversions**: involves converting the value of the operand expression of a cast operator.
- Casting between primitive data types (e.g. boolean to integer) and reference types (e.g. class to interface).
- The reference literal **`null`** can be cast to any reference type.

```
long l = (long) 10;    // Widening primitive conversion: long <- int
int i = (int) l;       // (1) Narrowing primitive conversion: int <- long
Object obj = (Object) "7Up"; // Widening ref conversion: Object <- String
String str = (String) obj;   // (2) Narrowing ref conversion: String <-
Object
Integer iRef = (Integer) i;  // Boxing: Integer <- int
i = (int) iRef;              // Unboxing: int <- Integer
```

Type Conversion Contexts

There are mainly four conversion contexts -

- **Numeric Promotion**: results in conversions being applied to the operands to convert them to permissible types (as numeric operators allow only operands of certain types).
- Unary numeric promotion results in an operand value that is either `int` or wider.
- Binary numeric promotion results in the wider type (i.e. both the operands are converted into the wider type if they are not of the same type).

Type Conversion Contexts: Summary

Conversion categories	Conversion contexts			
	Assignment	**Method invocation**	**Casting**	**Numeric promotion**
Widening / narrowing *primitive* conversions	Widening Narrowing for *constant expressions* of non-`long` integral type, with optional boxing	Widening	Both	Widening
Widening / narrowing *reference* conversions	Widening	Widening	Both, followed by optional unchecked conversion	Not applicable
Boxing / unboxing conversions	Unboxing, followed by optional widening *primitive* conversion Boxing, followed by optional widening *reference* conversion	Unboxing, followed by optional widening *primitive* conversion Boxing, followed by optional widening *reference* conversion	Both	Unboxing, followed by optional widening *primitive* conversion

Operator Summary

Array element access, member access, method invocation	`[expression] . (args)`
Unary postfix operators	`expression++ expression--`
Unary prefix operators	`~ ! ++expression --expression +expression -expression`
Unary prefix creation and cast	`new (type)`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >> >>>`
Relational	`< <= > >= instanceof`
Equality	`== !=`
Bitwise/logical AND	`&`
Bitwise/logical XOR	`^`
Bitwise/logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Conditional	`?:`
Arrow operator	`->`
Assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`

Precedence and Associativity

The evaluation order of the operators are determined by the precedence and associativity rules.

Precedence

- Precedence rules are used to determine which operator should be applied first if there are ***two or more operators with different precedence*** in the expression.

e.g. `2 + 3 * 4` ⇒ `2 + (3 * 4)` `//` result is 14
`//` since * has higher precedence than +

Precedence and Associativity

The evaluation order of the operators are determined by the precedence and associativity rules.

Associativity

- Associativity rules are used to determine which operator should be applied first if there are ***two or more operators with same precedence*** in the expression.

e.g. `7 - 4 + 2` ⇒ `(7 - 4) + 2` `//` result is 5
`//` since **+** and - has same precedence and left associativity

Evaluation Order of Operands

Left-Hand Operand Evaluation First

- The left-hand operand of a binary operator is fully evaluated before the right-hand operand is evaluated.

```
int b = 10;
System.out.println((b=3) + 3); // prints 6 not 13
```

Evaluation Order of Operands

Operand Evaluation before Operation Execution

- Java guarantees that all operands of an operator are fully evaluated before the actual operation is performed.

```
import static java.lang.System.out;

public class EvalOrder{
  public static void main(String[] args){

    int j = 2;
    out.println("Evaluation order of operands:");
    out.println(eval(j++, " + ") + eval(j++, " * ") * eval(j, "\n"));    //
(1)

    int i = 1;
    out.println("Evaluation order of arguments:");
    add3(eval(i++, ", "), eval(i++, ", "), eval(i, "\n")); // (2) Three
arguments.
  }

  public static int eval(int operand, String str) {        // (3)
    out.print(operand + str);       // Print int operand and String str.
    return operand;                 // Return int operand.
  }

  public static void add3(int operand1, int operand2, int operand3) {    //
(4)
    out.print(operand1 + operand2 + operand3);
  }
}
```

Evaluation Order of Operands

Left-to-Right Evaluation of Arguments Lists

- In a method invocation, each argument expression in the argument list is fully evaluated before any argument expression to its right..

```
import static java.lang.System.out;

public class EvalOrder{
  public static void main(String[] args){

    int j = 2;
    out.println("Evaluation order of operands:");
    out.println(eval(j++, " + ") + eval(j++, " * ") * eval(j, "\n"));    //
(1)

    int i = 1;
    out.println("Evaluation order of arguments:");
    add3(eval(i++, ", "), eval(i++, ", "), eval(i, "\n")); // (2) Three
arguments.
  }

  public static int eval(int operand, String str) {        // (3)
    out.print(operand + str);       // Print int operand and String str.
    return operand;                 // Return int operand.
  }

  public static void add3(int operand1, int operand2, int operand3) {    //
(4)
    out.print(operand1 + operand2 + operand3);
  }
}
```

Next lecture -

Operators & Expressions and Control Flow