


ADVANCED OBJECT ORIENTED PROGRAMMING

Engr. Anees Ahmed Soomro

Assistant Professor

CS QUEST Nawabshah

<https://anees-soomro.neocities.org>

- 
- 1) Mathematical functions and constants
 - 2) Type Conversion and Casting,
 - 3) Operators
 - 4) Operators precedence

Mathematical functions and constants, Expressions

- The final class `Math` defines a set of static methods to support common mathematical functions, including functions for rounding numbers, trigonometry, pseudo random numbers, finding maximum and minimum of two numbers, calculating logarithms and exponentiation.
- The `Math` class cannot be instantiated. Only the class name `Math` can be used to invoke the static methods.
- The final class `Math` provides constants to represent the value of e , the base of the natural logarithms, and the value (`pi`), the ratio of the circumference of a circle to its diameter:
`Math.E` ,`Math.PI`.

Miscellaneous Rounding Functions

`static int abs(int i)`

`static long abs(long l)`

`static float abs(float f)`

`static double abs(double d)`

- The overloaded method `abs()` returns the absolute value of the argument. For a non-negative argument, the argument is returned. For a negative argument, negation of the argument is returned.

`static int min(int a, int b)`

`static long min(long a, long b)`

`static float min(float a, float b)`

`static double min(double a, double b)`

- The overloaded method `min()` returns the smaller of the two values `a` and `b` for any numeric type.

`static int max(int a, int b)`

`static long max(long a, long b)`

`static float max(float a, float b)`

`static double max(double a, double b)`

- The overloaded method `max()` returns the greater of the two values `a` and `b` for any numeric type.

- The following code illustrates the use of these methods from the Math class:

```
long ll = Math.abs(2010L); // 2010L
double dd = Math.abs(-Math.PI); // 3.141592653589793
double d1 = Math.min(Math.PI, Math.E); // 2.718281828459045
long m1 = Math.max(1984L, 2010L); // 2010L
int i1 = (int) Math.max(3.0, 4); // Cast required.
```

Note the cast required in the last example. The method with the signature `max(double, double)` is executed, with implicit conversion of the `int` argument to a `double`.

- Since this method returns a `double`, it must be explicitly cast to an `int`.

```
static double ceil(double d)
```

- The method `ceil()` returns the smallest `double` value that is greater than or equal to the argument `d`, and is equal to a mathematical integer.

```
static double floor(double d)
```

- The method `floor()` returns the largest `double` value that is less than or equal to the argument `d`, and is equal to a mathematical integer.

```
static int round(float f) static long round(double d)
```

- The overloaded method `round ()` returns the integer closest to the argument.
- This is equivalent to adding 0.5 to the argument, taking floor of the result, and casting it to the return type. This is not same as rounding to a specific number of decimal places, as name of the method might suggest.

- If fractional part of a positive argument is less than 0.5, then result returned is same as `Math.floor()`.
- If the fractional part of a positive argument is greater than or equal to 0.5, then the result returned is the same as `Math.ceil()`.
- If fractional part of negative argument is less than or equal to 0.5, then result returned is same as `Math.ceil()`.
- If fractional part of a negative argument is greater than 0.5, then result returned is the same as `Math.floor()`.
- It is important to note result obtained on negative arguments, keeping in mind that a negative number whose absolute value is less than that of another negative number, is actually greater than the other number (e.g., -3.2 is greater than -4.7). Compare also the results returned by these methods, shown in [Table 10.1](#).
- `double upPI = Math.ceil(Math.PI); // 4.0`
- `double downPI = Math.floor(Math.PI); // 3.0`
- `long roundPI = Math.round(Math.PI); // 3L`
- `double upNegPI = Math.ceil(-Math.PI); // -3.0`
- `double downNegPI = Math.floor(-Math.PI); // -4.0`
- `long roundNegPI = Math.round(-Math.PI); // -3L`



Arg:	7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	7.9	8.0
ceil:	7.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0
floor:	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
round:	7	7	7	7	7	8	8	8	8	8	8
Arg:	-7.0	-7.1	-7.2	-7.3	-7.4	-7.5	-7.6	-7.7	-7.8	-7.9	-8.0
ceil:	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-8.0
floor:	-7.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0
round:	-7	-7	-7	-7	-7	-7	-8	-8	-8	-8	-8

Exponential Functions

static double pow(double d1, double d2)

- The method pow() returns the value of d1 raised to the power of d2 (i.e., $d1^{d2}$).

static double exp(double d)

- The method exp() returns the exponential number e raised to the power of d (i.e., e^d).

static double log(double d)

- The method log() returns the natural logarithm (base e) of d (i.e., $\log_e d$).

static double sqrt(double d)

- The method sqrt() returns the square root of d (i.e., $d^{0.5}$).
- For a NaN or a negative argument, the result is a NaN (see [Section 3.5](#), p. 52).
- Some examples of exponential functions:
- `double r = Math.pow(2.0, 4.0); // 16.0`
- `double v = Math.exp(2.0); // 7.38905609893065`
- `double l = Math.log(Math.E); // 0.99999999999999981`
- `double c = Math.sqrt(3.0*3.0 + 4.0*4.0); // 5.0`

Trigonometry Functions

- static double `sin(double d)`

The method `sin()` returns the trigonometric sine of an angle `d` specified in radians.

- static double `cos(double d)`

The method `cos()` returns the trigonometric cosine of an angle `d` specified in radians.

- static double `tan(double d)`

The method `tan()` returns the trigonometric tangent of an angle `d` specified in radians.

- static double `toRadians(double degrees)`

The method `toRadians()` converts an angle in degrees to its approximation in radians.

- static double `toDegrees(double radians)`

The method `toDegrees()` converts an angle in radians to its approximation in degrees.

- Some examples of trigonometry functions:

- `double deg1 = Math.toDegrees(Math.PI/4.0); // 45 degrees`

- `double deg2 = Math.toDegrees(Math.PI/2.0); // 90 degrees`

- `double rad1 = Math.toRadians(deg1); // 0.7853981633974483`

- `double rad2 = Math.toRadians(deg2); // 1.5707963267948966`

- `double r1 = Math.sin(Math.PI/2.0); // 1.0`

- `double r2 = Math.sin(Math.PI*2); // -2.4492935982947064E-16 (0.0)`

- `double r3 = Math.cos(Math.PI); // -1.0`

- `double r4 = Math.cos(Math.toRadians(360.0)); // 1.0`

- `double r5 = Math.tan(Math.toRadians(90.0)); // 1.633123935319537E16 (infinity)`

- `double r6 = Math.tan(Math.toRadians(45.0)); // 0.9999999999999999 (1.0)`

Expected mathematical values are shown in parentheses.

Pseudorandom Number Generator

`static double random()`

- The method `random()` returns a random number greater than or equal to 0.0 and less than 1.0, where the value is selected randomly from the range according to a uniform distribution.
- An example of using the pseudorandom number generator is as follows:
- `for (int i = 0; i < 10; i++)`
- `System.out.println((int)(Math.random()*10)); // int values in range [0 .. 9].`
- The loop will generate a run of ten pseudorandom integers between 0 (inclusive) and 10 (exclusive).

Arithmetic Operators: *, /, %, +, -

- The arithmetic operators are used to construct mathematical expressions as in algebra. Their operands are of numeric type (which includes the `char` type).

Arithmetic Operator Precedence and Associativity

In [Table 3.3](#), the precedence of the operators is in decreasing order, starting from the top row, which has the highest precedence. Unary subtraction has higher precedence than multiplication.

The operators in the same row have the same precedence. Binary multiplication, division, and remainder operators have the same precedence. The unary operators have right associativity, and the binary operators have left associativity.

Evaluation Order in Arithmetic Expressions

Table 3.3. Arithmetic Operators

Unary	+	Addition	-	Subtraction		
Binary	*	Multiplication	/	Division	%	Remainder
	+	Addition	-	Subtraction		

Java guarantees that operands are fully evaluated from left to right before an arithmetic binary operator is applied. Of course, if evaluation of an operand causes an exception, the subsequent operands will not be evaluated.

In the expression $a + b * c$, the operand a will always be fully evaluated before the operand b , which will always be fully evaluated before the operand c . However, the multiplication operator $*$ will be applied before the addition operator $+$, respecting the precedence rules. Note that a , b , and c can be arbitrary arithmetic expressions that have been determined to be the operands of the operators.

Table 3.4. Examples of Arithmetic Expression Evaluation

Arithmetic Expression	Evaluation	Result When Printed
$3 + 2 - 1$	$((3 + 2) - 1)$	4
$2 + 6 * 7$	$(2 + (6 * 7))$	44
$-5+7- -6$	$(((-5)+7)-(-6))$	8
$2+4/5$	$(2+(4/5))$	2
$13 \% 5$	$(13 \% 5)$	3
$11.5 \% 2.5$	$(11.5 \% 2.5)$	1.5
$10 / 0$	ArithmeticException	
$2+4.0/5$	$(2.0+(4.0/5.0))$	2.8
$4.0 / 0.0$	$(4.0 / 0.0)$	Infinity
$-4.0 / 0.0$	$((-4.0) / 0.0)$	-Infinity
$0.0 / 0.0$	$(0.0 / 0.0)$	

Numeric Promotions in Arithmetic Expressions

- Unary numeric promotion is applied to the single operand of unary arithmetic operators - and +.
- In other words, when a unary operator is applied to an operand of byte, short or char type, the operand is first promoted to a value of type int, with the evaluation resulting in an int value.
- If the conditions for implicit narrowing conversion are not fulfilled (p. 48), assigning the int result to a variable of these types will require an explicit cast.
- This is demonstrated by the following example, where the byte operand b is promoted to an int in the expression (-b):
- `byte b = 3; // int literal in range. Implicit narrowing.`
- `b = (byte) -b; // Explicit narrowing on assignment required.`
- Binary numeric promotion is applied to operands of binary arithmetic operators. Its application leads to automatic type promotion for the operands. The result is of the promoted type, which is always type int or wider. For the expression at (1) in [Example 3.1](#), numeric promotions proceed as shown in [Figure 3.3](#).
- Note the integer division performed in evaluating the subexpression (c / s).

Numeric Promotion in Arithmetic Expressions

```
public class NumPromotion {  
    public static void main(String[] args) {  
        byte b = 32; char c = 'z'; // Unicode value 122 (\u007a)  
        short s = 256;  
        int i = 10000;  
        float f = 3.5F;  
        double d = 0.5;  
        double v = (d * i) + (f * - b) - (c / s); // (1) 4888.0D  
        System.out.println("Value of v: " + v);  
    }  
}
```

Output from the program:

Value of v: 4888.0

Type Conversion and Casting

Conversions

- In this section we discuss the different kinds of type conversions and list the contexts in which these can occur. Some type conversions must be explicitly stated in the program, while others are done implicitly. Some type conversions can be checked at compile time to guarantee their validity at runtime, while others will require an extra check at runtime.

Unary Cast Operator: (type)

- Java, being a strongly typed language, checks for type compatibility (i.e., checks if a type can substitute for another type in a given context) at compile time. However, some checks are only possible at runtime (for example, which type of object a reference actually denotes during execution). In cases where an operator would have incompatible operands (for example, assigning a double to an int), Java demands that a cast be used to explicitly indicate the type conversion.
- The cast construct has the following syntax:

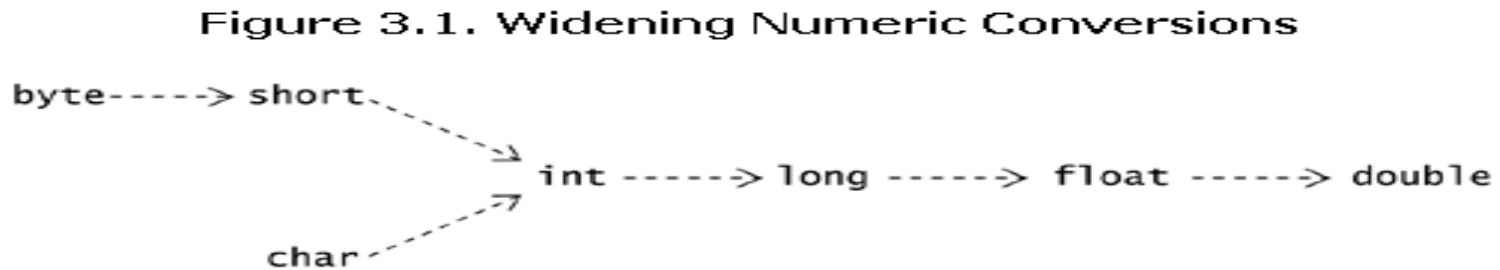
(<type>) <expression>

- The cast (<type>) is applied to the value of the <expression>. At runtime, a cast results in a new value of <type>, which best represents the value of the <expression> in the old type. We use the term casting to mean applying the cast operator for explicit type conversion.
- **Casting can be applied to primitive values as well as references. Casting between primitive data types and reference types is not permitted. Boolean values cannot be cast to other data values, and vice versa.**
- The reference literal null can be cast to any reference type.

Narrowing and Widening Conversions

- For the primitive data types, the value of a narrower data type can be converted to a value of a broader data type without loss of information. This is called a widening primitive conversion.
- The conversions shown are transitive. For example, an int can be directly converted to a double without first having to convert it to a long and a float.

Figure 3.1. Widening Numeric Conversions



- Converting from a broader data type to a narrower data type is called a narrowing primitive conversion, which can result in loss of magnitude information. Note that all conversions between char and the two integer types byte and short are considered narrowing conversions: the reason being that the conversions between the unsigned type char and the signed types byte or short can result in loss of information.
- Widening and narrowing conversions are also defined for reference types. Conversions up the inheritance hierarchy are called widening reference conversions (**a.k.a. upcasting**). Conversions down the inheritance hierarchy are called narrowing reference conversions (**a.k.a. downcasting**).

Both narrowing and widening conversions can be either explicit (requiring a cast) or implicit. Widening conversions are usually done implicitly, whereas narrowing conversions typically require a cast. It is not illegal to use a cast for a widening conversion. However, the compiler will flag any conversions that require a cast.

Numeric Promotions

- Numeric operators only allow operands of certain types.
- Numeric promotion is implicitly applied on the operands to convert them to permissible types. Distinction is made between unary and binary numeric promotion.

Unary Numeric Promotion

- Unary numeric promotion states that
- If the single operand of the operator has a type narrower than int, it is converted to int by an implicit widening primitive conversion; otherwise, it is not converted.
- In other words, unary numeric promotion converts operands of byte, short, and char to int by applying an implicit widening conversion, but operands of other numeric types are not affected.
- Unary numeric promotion is applied in the following contexts:
- operand of the unary arithmetic operators + and –

- operand of the unary integer bitwise complement operator `~`
- during array creation; for example, `new int[20]`, where the dimension expression (in this case `20`) must evaluate to an `int` value
- indexing array elements; for example, `table['a']`, where the index expression (in this case `'a'`) must evaluate to an `int` value
- individual operands of the shift operators `<<`, `>>` and `>>>`

Binary Numeric Promotion

- Binary numeric promotion implicitly applies appropriate widening primitive conversions so that a pair of operands have the broadest numeric type of the two, which is always at least `int`. Given `T` to be the broadest numeric type of the two operands, the operands are promoted as follows under binary numeric promotion.
- If `T` is broader than `int`, both operands are converted to `T`; otherwise, both operands are converted to `int`.
- This means that `byte`, `short`, and `char` are always converted to `int` at least.
- Binary numeric promotion is applied in the following contexts:
 - operands of the arithmetic operators `*`, `/`, `%`, `+`, and `-`
 - operands of the relational operators `<`, `<=`, `>`, and `>=`
 - operands of the numerical equality operators `==` and `!=`
 - operands of the integer bitwise operators `&`, `^`, and `|`

Type Conversion Contexts

Type conversions can occur in the following contexts: assignments involving primitive data types and reference types. method invocation involving parameters of primitive data types and reference types. arithmetic expression evaluation involving numeric types. String concatenation involving objects of class String and other data types

Simple Assignment Operator =

- The assignment statement has the following syntax: <variable> = <expression>
- which can be read as "the destination, <variable>, gets the value of the source, <expression>".
- The previous value of the destination variable is overwritten by the assignment operator =.
- The destination <variable> and the source <expression> must be type compatible. The destination variable must also have been declared. Since variables can store either primitive data values or object references, <expression> evaluates to either a primitive data value or an object reference.

Assigning Primitive Values

- The following examples illustrate assignment of primitive values:

```
int j, k;
```

```
j = 10; // j gets the value 10.
```

```
j = 5; // j gets the value 5. Previous value is overwritten.
```

```
k = j; // k gets the value 5. The assignment operator has the lowest precedence, allowing the expression on the right-hand side to be evaluated before assignment.
```

```
int i;
```

```
i = 5; // i gets the value 5.
```

```
i = i + 1; // i gets the value 6. + has higher precedence than =.
```

```
i = 20 - i * 2; // i gets the value 8: (20 - (i * 2))
```

Multiple Assignments

- The assignment statement is an expression statement, which means that application of the binary assignment operator returns the value of the expression on the right-hand side.

```
int j, k;
```

```
j = 10;      // j gets the value 10 which is returned
```

```
k = j;      // k gets the value of j, which is 10, and this value is returned
```

- The last two assignments can be written as multiple assignments, illustrating the right associativity of the assignment operator.

```
k = j = 10;  // (k = (j = 10)) Multiple assignments are equally valid with references.
```

```
Pizza pizzaOne, pizzaTwo; pizzaOne = pizzaTwo = new Pizza("Supreme");
```

```
// Aliases. The following example shows the effect of operand evaluation order:
```

```
int[] a = {10, 20, 30, 40, 50}; // an array of int
```

```
int index = 4;
```

```
a[index] = index = 2;
```

- What is the value of index, and which array element a[index] is assigned a value in the multiple assignment statement? The evaluation proceeds as follows:

```
a[index] = index = 2;
```

```
a[4] = index = 2;
```

```
a[4] = (index = 2);    // index gets the value 2. = is right associative.
```

```
a[4] = 2;             // The value of a[4] is changed from 50 to 2.
```

Numeric Type Conversions on Assignment

- If the destination and source have the same type in an assignment, then, obviously, the source and the destination are type compatible, and the source value need not be converted. Otherwise, if a widening primitive conversion is permissible, then the widening conversion is applied implicitly, that is, the source type is promoted to the destination type in an assignment context.

// Implicit Widening Primitive Conversions

```
int smallOne = 1234;
```

```
long bigOne = 2000; // Implicit widening: int to long.
```

```
double largeOne = bigOne; // Implicit widening: long to double.
```

```
double hugeOne = (double) bigOne; // Cast redundant but allowed.
```

- Integer values widened to floating-point values can result in loss of precision. Precision relates to the number of significant bits in the value, and must not be confused with magnitude, which relates how big a value can be represented. In the next example, the precision of the least significant bits of the long value may be lost when converting to a float value.

```
long bigInteger = 98765432112345678L;
```

```
float realNo = bigInteger; // Widening but loss of precision: 9.8765436E16
```

Additionally, implicit narrowing primitive conversions on assignment can occur in cases where all of the following conditions are fulfilled:

- the source is a constant expression of either byte, short, char, or int type
- the destination type is either byte, short, or char type
- the value of the source is determined to be in the range of the destination type at compile time

- // Above conditions fulfilled for implicit narrowing primitive conversions.
- `short s1 = 10;` // int value in range.
- `short s2 = 'a';` // char value in range.
- `char c1 = 32;` // int value in range.
- `char c2 = (byte)35;` // byte value in range. int value in range, without cast.
- `byte b1 = 40;` // int value in range.
- `byte b2 = (short)40;` // short value in range.
- int value in range, without cast.
- `final int i1 = 20;`
- `byte b3 = i1;` // final value of i1 in range Above conditions not fulfilled for implicit //narrowing primitive conversions Explicit cast required.
- `int i2 = -20;`
- `final int i3 = i2;`
- `final int i4 = 200;`
- `short s3 = (short) i2;` // Not constant expression.
- `char c3 = (char) i3;` // final value of i3 not determinable.
- `char c4 = (char) i2;` // Not constant expression.
- `byte b4 = (byte) 128;` // int value not in range.
- `byte b5 = (byte) i4;` // final value of i4 not in range.

- All other narrowing primitive conversions will produce a compile-time error on assignment, and will explicitly require a cast.
- Floating-point values are truncated when converted to integral values.

```
// Explicit narrowing primitive conversions requiring cast.
```

```
// The value is truncated to fit the size of the destination type.
```

```
float huge = (float) 1.7976931348623157d; // double to float.
```

```
long giant = (long) 4415961481999.03D; // (1) double to long.
```

```
int big = (int) giant; // (2) long to int.
```

```
short small = (short) big; // (3) int to short.
```

```
byte minute = (byte) small; // (4) short to byte.
```

```
char symbol = (char) 112.5F; // (5) float to char.
```

Casting

- Casting an object reference is necessary when the compiler can't tell that the actual object is of the needed type. The cast creates a runtime check to make sure. There are two cases where a cast is required:

- To cast down the inheritance hierarchy. For example,

```
Object ob = "abc"; //No cast needed to assign up hierarchy.
```

```
String s = ob; // BAD. Requires a cast
```

- String s = (String)ob; // Required.
- All Strings are Objects, but not all Objects are Strings.

- [an error occurred while processing this directive] To cast from an interface type to a class type. (String implements Comparable)
- `Comparable com = "def"; // No cast to assign to implemented interface String s = com; // BAD.`
- Requires a cast. `String s = (String)com; // Required.`
- All Strings are Comparables, but not all Comparables are Strings.`