Software Development (cs2500)

Lecture 6: Fundamental Data Types & Operations

M. R. C. van Dongen

October 4, 2013

Basic Types

		Range		Whole	Size in
Type	Description	Smallest	Largest	Number	Bytes
`boolean`	truth value	`true`	`false`	?	?
`char`	character	`'\u0000'`	`'\uFFFF'`	?	4
`byte`	byte	-2^7	2^7-1	+	1
`short`	short integer	-2^{15}	$2^{15}-1$	+	2
`int`	integer	-2^{31}	$2^{31}-1$	+	4
`long`	long integer	-2^{63}	$2^{63}-1$	+	8
`float`	FP nr	-10^{38}	10^{38}	−	8
`double`	double precision FP nr	-10^{308}	10^{308}	−	16

Literals

String Literal

Java

```java
final String QUESTION = "What’s the answer?";
final int ANSWER = 42;
```

Literals

int Literal

Java

```
final String QUESTION = "What's the answer?";
final int ANSWER = 42;
```

Primitive Integer Literals

- Integer literals are usually represented as decimal numbers.

Java

```
short s = 100;
int   i = 0;
long  l = -100;
```

- This is the default representation.
  - `Java` assumes that each such literal is an `int`.
- However, the value must be in the right range:
  - `byte s = 128` is not allowed.

Primitive `long` Literals

- Occasionally, you need to write a `long` literal.
- Adding an ‘`l`’ or ‘`L`’ at the end turns the literal into a `long`.

Java

```
    long long1 = 2147483647;  // Largest possible int.
    long long2 = 2147483648;  // Too large: not allowed.
    long long3 = 21474836481; // Also too large.
    long long4 = 2147483648l; // Allowed but not clear.
```

Primitive long Literals

- Occasionally, you need to write a long literal.
- Adding an ‘l’ or ‘L’ at the end turns the literal into a long.

Java

```
long long1 = 2147483647;  // Largest possible int.
long long2 = 2147483648;  // Too large: not allowed.
long long3 = 21474836481; // Also too large.
long long4 = 2147483648l; // Allowed but not clear.
```

Primitive `long` Literals

- Occasionally, you need to write a `long` literal.
- Adding an ‘`l`’ or ‘`L`’ at the end turns the literal into a `long`.

Java

```
    long long1 = 2147483647;  // Largest possible int.
    long long2 = 2147483648;  // Too large: not allowed.
    long long3 = 21474836481; // Also too large.
    long long4 = 2147483648l; // Allowed but not clear.
```

Primitive long Literals

- Occasionally, you need to write a long literal.
- Adding an ‘l’ or ‘L’ at the end turns the literal into a long.

Java

```
long long1 = 2147483647;  // Largest possible int.
long long2 = 2147483648;  // Too large: not allowed.
long long3 = 21474836481; // Also too large.
long long4 = 2147483648l; // Allowed but not clear.
```

Primitive long Literals

- Occasionally, you need to write a long literal.
- Adding an ‘l’ or ‘L’ at the end turns the literal into a long.

Java

```
long long1 = 2147483647;  // Largest possible int.
long long2 = 2147483648;  // Too large: not allowed.
long long3 = 21474836481; // Also too large.
long long4 = 2147483648l; // Allowed but not clear.
```

Primitive long Literals

- Occasionally, you need to write a long literal.
- Adding an ‘l’ or ‘L’ at the end turns the literal into a long.

Java

```java
long long1 = 2147483647;  // Largest possible int.
long long2 = 2147483648;  // Too large: not allowed.
long long3 = 2147483648l; // Also too large.
long long4 = 2147483648l; // Allowed but not clear.
```

Primitive long Literals

- Occasionally, you need to write a long literal.
- Adding an 'l' or 'L' at the end turns the literal into a long.

Java

```
long long1 = 2147483647;  // Largest possible int.
long long2 = 2147483648;  // Too large: not allowed.
long long3 = 21474836481; // Also too large.
long long4 = 2147483648l; // Allowed but not clear.
long long5 = 2147483648L; // Perfect!
```

Primitive long Literals

- Occasionally, you need to write a long literal.
- Adding an ‘l’ or ‘L’ at the end turns the literal into a long.

Java

```
long long1 = 2147483647;  // Largest possible int.
long long2 = 2147483648;  // Too large: not allowed.
long long3 = 2147483641; // Also too large.
long long4 = 2147483648l; // Allowed but not clear.
long long5 = 2147483648L; // Perfect!
```

Other Base Systems: Not Examinable

- Integral literals may also be written in octal and hexadecimal.
- Octal literals start with a zero (sigh):
  - ‘`022`’ corresponds to ‘`18`.’
- Hexadecimal literals start with the string ‘`0x`’:
  - ‘`0x12`’ corresponds to ‘`18`.’
- Starting hexadecimal literals with ‘`0X`’ is also allowed.

Floating Point Literals

- By default, all floating point literals are `doubles`.
- You write floating point literals like this:
  - ‘⟨`sign option`⟩⟨`digit sequence`⟩`.`⟨`digit sequence`⟩.’
  - ‘⟨`sign option`⟩⟨`digit sequence`⟩`.`’ or ‘⟨`sign option`⟩`.`⟨`digit sequence`⟩’
  - ‘⟨`base`⟩⟨`exponent`⟩’, where ⟨`base`⟩ is given by

    ‘⟨`sign option`⟩⟨`digit sequence`⟩`.`⟨`digit sequence`⟩’,

    and ⟨`exponent`⟩ is given by

    ‘⟨`E or e`⟩⟨`sign option`⟩⟨`digit sequence`⟩’.
  - Variations of scientific notation are also possible.

Explicit `float`/`double` Literals

- Adding ‘`f`’ or ‘`F`’ at the end turns literal into a ‘`float`.’
- If you need a `float` the extra letter is required.
- Adding ‘`d`’ or ‘`D`’ at the end “turns” literal into a ‘`double`.’

Java

```
double d1 = 1.0E10;     // Grand.
double d2 = -1.0E-10D; // Grand.
double d3 = -.1;        // Grand.
float  f1 = 1.0;        // Not allowed.
float  f2 = 1.00F;      // Grand.
float  f3 = -1.0E-10F; // Grand.
```

Explicit `float`/`double` Literals

- Adding ‘f’ or ‘F’ at the end turns literal into a ‘`float`.’
- If you need a `float` the extra letter is required.
- Adding ‘d’ or ‘D’ at the end “turns” literal into a ‘`double`.’

Java

```
double d1 = 1.0E10;    // Grand.
double d2 = -1.0E-10D; // Grand.
double d3 = -.1;       // Grand.
float  f1 = 1.0;       // Not allowed.
float  f2 = 1.00F;     // Grand.
float  f3 = -1.0E-10F; // Grand.
```

Explicit `float`/`double` Literals

- Adding ‘`f`’ or ‘`F`’ at the end turns literal into a ‘`float`.’
- If you need a `float` the extra letter is required.
- Adding ‘`d`’ or ‘`D`’ at the end “turns” literal into a ‘`double`.’

Java

```
double d1 = 1.0E10;     // Grand.
double d2 = -1.0E-10D; // Grand.
double d3 = -.1;        // Grand.
float  f1 = 1.0;        // Not allowed.
float  f2 = 1.00F;      // Grand.
float  f3 = -1.0E-10F; // Grand.
```

Explicit `float`/`double` Literals

- Adding ‘`f`’ or ‘`F`’ at the end turns literal into a ‘`float`.’
- If you need a `float` the extra letter is required.
- Adding ‘`d`’ or ‘`D`’ at the end “turns” literal into a ‘`double`.’

Java

```
double d1 = 1.0E10;    // Grand.
double d2 = -1.0E-10D; // Grand.
double d3 = -.1;       // Grand.
float  f1 = 1.0;       // Not allowed.
float  f2 = 1.00F;     // Grand.
float  f3 = -1.0E-10F; // Grand.
```

Explicit `float`/`double` Literals

- Adding '`f`' or '`F`' at the end turns literal into a '`float`.'
- If you need a `float` the extra letter is required.
- Adding '`d`' or '`D`' at the end "turns" literal into a '`double`.'

Java

```
double d1 = 1.0E10;     // Grand.
double d2 = -1.0E-10D; // Grand.
double d3 = -.1;        // Grand.
float  f1 = 1.0;        // Not allowed.
float  f2 = 1.00F;      // Grand.
float  f3 = -1.0E-10F; // Grand.
```

Explicit float/double Literals

- Adding ‘f’ or ‘F’ at the end turns literal into a ‘float.’
- If you need a float the extra letter is required.
- Adding ‘d’ or ‘D’ at the end “turns” literal into a ‘double.’

Java

```
double d1 = 1.0E10;     // Grand.
double d2 = -1.0E-10D; // Grand.
double d3 = -.1;        // Grand.
float  f1 = 1.0;        // Not allowed.
float  f2 = 1.00F;      // Grand.
float  f3 = -1.0E-10F; // Grand.
```

Explicit `float`/`double` Literals

- Adding ‘`f`’ or ‘`F`’ at the end turns literal into a ‘`float`.’
- If you need a `float` the extra letter is required.
- Adding ‘`d`’ or ‘`D`’ at the end “turns” literal into a ‘`double`.’

Java

```
double d1 = 1.0E10;     // Grand.
double d2 = -1.0E-10D; // Grand.
double d3 = -.1;        // Grand.
float  f1 = 1.0;        // Not allowed.
float  f2 = 1.00F;      // Grand.
float  f3 = -1.0E-10F; // Grand.
```

Character Literals

- Character literals are always written inside single quotes.
- There are three main classes of character literals:

Normal characters:	Unicode characters: `'a'`, `'B'`, `'ñ'`, …
Escape sequences:	`'\n'`, `'\t'`, `'\"'`, `'\''`, `'\\'`, …
Unicode escapes:	`'\u`⟨hexadecimal number⟩`'`.

Representation of Integral Values

- Java's represents integral types as two's complement integers.
- Two's complement supports *signed* and *unsigned* operations.
- Java only supports signed integers.

One’s Complement

- *One’s complement* representation flips bits.
- Flipping a bit, b, means turning it into its complement, $1 - b$.
- This turns a 1 into a 0 and a 0 into a 1.

Two’s Complement

- Java represents n-bit integers in *two’s complement* format.

  Non-negative: 0 followed by $n - 1$ more bits.
  Negative: Take absolute value, one’s complement, & add 1.
- Largest possible number is $0111111 \cdots 1$.
  - This bit sequence represents the number $2^{n-1} - 1$.
- A bit sequence represents a negative number if it starts with 1.
- Smallest possible number is $1000000 \cdots 0$.
  - This bit sequence represents the number -2^{n-1}.
- In total there are $2^{n-1} - 1 + 2^{n-1} + 1 = 2^n$ values.

Two-s Complement Representation of -1

- First take the representation of absolute value of -1:

Representation of `abs( -1 )`			
B_3	B_2	B_1	B_0
00000000	00000000	00000000	00000001

Two-s Complement Representation of `-1`

- First take the representation of absolute value of -1:

Representation of `abs( -1 )`			
B_3	B_2	B_1	B_0
00000000	00000000	00000000	00000001

- Next take the one's complement:

One's Complement			
B_3	B_2	B_1	B_0
11111111	11111111	11111111	11111110

Two-s Complement Representation of -1

- First take the representation of absolute value of -1:

Representation of abs(-1)			
B_3	B_2	B_1	B_0
00000000	00000000	00000000	00000001

- Next take the one's complement:

One's Complement			
B_3	B_2	B_1	B_0
11111111	11111111	11111111	11111110

- Finally, add 1:

Add 1			
B_3	B_2	B_1	B_0
11111111	11111111	11111111	11111111

Two-s Complement Representation of -3

- First take the representation of absolute value of -3.

Representation of `abs( -3 )`			
B_3	B_2	B_1	B_0
00000000	00000000	00000000	00000011

Two-s Complement Representation of -3

- First take the representation of absolute value of -3.

Representation of `abs( -3 )`			
B_3	B_2	B_1	B_0
00000000	00000000	00000000	00000011

- Next take the one's complement:

One's Complement			
B_3	B_2	B_1	B_0
11111111	11111111	11111111	11111100

Two-s Complement Representation of -3

- First take the representation of absolute value of -3.

Representation of `abs( -3 )`			
B_3	B_2	B_1	B_0
00000000	00000000	00000000	00000011

- Next take the one's complement:

One's Complement			
B_3	B_2	B_1	B_0
11111111	11111111	11111111	11111100

- Finally, add 1:

Add 1			
B_3	B_2	B_1	B_0
11111111	11111111	11111111	11111101

Two-s Complement Representation of -2^{n-1}

- First take the representation of absolute value of -2^{n-1}.

Representation of abs(-2^{n-1})			
B_3	B_2	B_1	B_0
10000000	00000000	00000000	00000000

Two-s Complement Representation of -2^{n-1}

- First take the representation of absolute value of -2^{n-1}.

Representation of abs(-2^{n-1})			
B_3	B_2	B_1	B_0
10000000	00000000	00000000	00000000

- Next take the one's complement:

One's Complement			
B_3	B_2	B_1	B_0
01111111	11111111	11111111	11111111

Two-s Complement Representation of -2^{n-1}

- First take the representation of absolute value of -2^{n-1}.

Representation of abs(-2^{n-1})			
B_3	B_2	B_1	B_0
10000000	00000000	00000000	00000000

- Next take the one's complement:

One's Complement			
B_3	B_2	B_1	B_0
01111111	11111111	11111111	11111111

- Finally, add 1:

Add 1			
B_3	B_2	B_1	B_0
10000000	00000000	00000000	00000000

Weird Innit?

- All primitive number types have a finite representation.
- You cannot represent whole numbers outside the range:

Don't Try This at Home

```
final int MAXIMUM = OX8FFFFFFF;
final int OVERFLOW = MAXIMUM + 1;
System.out.println( MAXIMUM );
System.out.println( OVERFLOW );
```

Weird Innit?

- All primitive number types have a finite representation.
- You cannot represent whole numbers outside the range:

Don't Try This at Home

```
final int MAXIMUM = 0X8FFFFFFF;
final int OVERFLOW = MAXIMUM + 1;
System.out.println( MAXIMUM ); // prints 2147483647
System.out.println( OVERFLOW );
```

Weird Innit?

- All primitive number types have a finite representation.
- You cannot represent whole numbers outside the range:

Don’t Try This at Home

```
final int MAXIMUM = 0X8FFFFFFF;
final int OVERFLOW = MAXIMUM + 1;
System.out.println( MAXIMUM ); // prints 2147483647
System.out.println( OVERFLOW ); // prints -1879048192
```

Weird Innit?

- The FP values are not continuous.
- Most FP computations result in rounding errors.

Don't Try This at Home

```
final double EPSILON = Double.MIN_VALUE;
final double NOTHING = EPSILON / 2.0;
final double LOSER = EPSILON * 1.5;

System.out.println( "EPSILON = " + EPSILON );
System.out.println( "NOTHING = " + NOTHING );
System.out.println( "LOSER = " + LOSER );
```

Weird Innit?

- The FP values are not continuous.
- Most FP computations result in rounding errors.

Don’t Try This at Home

```
final double EPSILON = Double.MIN_VALUE;
final double NOTHING = EPSILON / 2.0;
final double LOSER = EPSILON * 1.5;

System.out.println( "EPSILON = " + EPSILON ); // prints EPSILON = 4.9E-324
System.out.println( "NOTHING = " + NOTHING );
System.out.println( "LOSER = " + LOSER );
```

Weird Innit?

- The FP values are not continuous.
- Most FP computations result in rounding errors.

Don't Try This at Home

```
final double EPSILON = Double.MIN_VALUE;
final double NOTHING = EPSILON / 2.0;
final double LOSER = EPSILON * 1.5;

System.out.println( "EPSILON = " + EPSILON ); // prints EPSILON = 4.9E-324
System.out.println( "NOTHING = " + NOTHING ); // prints NOTHING = 0.0
System.out.println( "LOSER = " + LOSER );
```

Weird Innit?

- The FP values are not continuous.
- Most FP computations result in rounding errors.

Don't Try This at Home

```
final double EPSILON = Double.MIN_VALUE;
final double NOTHING = EPSILON / 2.0;
final double LOSER = EPSILON * 1.5;

System.out.println( "EPSILON = " + EPSILON ); // prints EPSILON = 4.9E-324
System.out.println( "NOTHING = " + NOTHING ); // prints NOTHING = 0.0
System.out.println( "LOSER = " + LOSER ); // prints LOSER = 1.0E-323
```

Counting

- Integer types are for counting.
- Using a FP number for counting things is an error.
- If numbers (may) have fractions you should use FP numbers.
- Use object types for large integers/fractions:

  `BigInteger` For integers:

  - `final BigInteger first = new BigInteger( "100000000000000" );`
  - `final BigInteger second = first.multiply( first );`

  `BigDecimal` For numbers with fractions:

  - `final BigDecimal first = new BigDecimal( "1234567890.0987654321" );`
  - `final BigDecimal second = first.multiply( first );`

What is the Output?

Java

```java
int a = 2 * 3 + 1;
int b = 2 * (3 + 1);
int c = (2 * 3) + 1;
System.out.println( a );
System.out.println( b );
System.out.println( c );
```

Why Evaluation Rules

- There are two reasons for having evaluation rules.
- They are related to “common sense” conventions.
- For example, when you write ‘`1 + 2 * 3`’
  - You expect `1 + (2 * 3)`,
  - Not `(1 + 2) * 3`.
- Likewise, when you write `1 - 2 - 3`
  - You expect `(1 - 2) - 3`,
  - Not `1 - (2 - 3)`.

Operators

- For simplicity we shall restrict our computations to arithmetic.
- Most of the time your programs use only a few operators:

Assignment:	=
Addition:	+
Subtraction:	-
Multiplication:	*
Division:	/
Remainder:	%
Plus:	unary +
Negation:	unary -

What do They Do?

- Most arithmetic operators are defined for integers and floats.
- The only operator not defined for floats is integer remainder: `%`.
- Dividing an integer by an integer is called *integer division*.
- The result is always an integer: the remainder is discarded.
  - `4 / 2` gives `2`; and `4 % 2` gives `0`;
  - `4 / 3` gives `1`; and `4 % 3` gives `1`.
- Dividing by zero is not allowed.
  - When a program attempts a division by `0`, you get a runtime error.
  - This also happens when the RHS of the remainder operation is `0`.

Integer Division and Remainder

- The remainder's sign is the same as that of the first operand.
- Let ⟨`lhs`⟩ be an integer and let ⟨`rhs`⟩ be a non-zero integer, then
  - ⟨`lhs`⟩ `/` ⟨`rhs`⟩ gives the integral part of dividing ⟨`lhs`⟩ by ⟨`rhs`⟩.
  - ⟨`lhs`⟩ `%` ⟨`rhs`⟩ gives the remainder of the division.
- In all cases we have the following equality:

$$\langle\texttt{lhs}\rangle = \left(\overbrace{(\langle\texttt{lhs}\rangle \mathbin{/} \langle\texttt{rhs}\rangle)}^{\text{quotient}} * \langle\texttt{rhs}\rangle\right) + \left(\overbrace{\langle\texttt{lhs}\rangle \mathbin{\%} \langle\texttt{rhs}\rangle}^{\textit{remainder}}\right) .$$

Examples

- 4 / 2 gives

Examples

- `4 / 2` gives 2,

Examples

- `4 / 2` gives 2, so `4 % 2` gives

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1,

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1,

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.
- 2 / 2 gives 1, so 2 % 2 gives 0.
- 1 / 2 gives

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.
- 2 / 2 gives 1, so 2 % 2 gives 0.
- 1 / 2 gives 0,

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.
- `1 / 2` gives 0, so `1 % 2` gives

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.
- `1 / 2` gives 0, so `1 % 2` gives 1.

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.
- 2 / 2 gives 1, so 2 % 2 gives 0.
- 1 / 2 gives 0, so 1 % 2 gives 1.
- 0 / 2 gives

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.
- `1 / 2` gives 0, so `1 % 2` gives 1.
- `0 / 2` gives 0,

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.
- `1 / 2` gives 0, so `1 % 2` gives 1.
- `0 / 2` gives 0, so `0 % 2` gives

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.
- `1 / 2` gives 0, so `1 % 2` gives 1.
- `0 / 2` gives 0, so `0 % 2` gives 0.

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.
- `1 / 2` gives 0, so `1 % 2` gives 1.
- `0 / 2` gives 0, so `0 % 2` gives 0.
- `7 / 3` gives

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.
- 2 / 2 gives 1, so 2 % 2 gives 0.
- 1 / 2 gives 0, so 1 % 2 gives 1.
- 0 / 2 gives 0, so 0 % 2 gives 0.
- 7 / 3 gives 2,

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.
- `1 / 2` gives 0, so `1 % 2` gives 1.
- `0 / 2` gives 0, so `0 % 2` gives 0.
- `7 / 3` gives 2, so `7 % 3` gives

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.
- 2 / 2 gives 1, so 2 % 2 gives 0.
- 1 / 2 gives 0, so 1 % 2 gives 1.
- 0 / 2 gives 0, so 0 % 2 gives 0.
- 7 / 3 gives 2, so 7 % 3 gives 1.

Examples

- `4 / 2` gives 2, so `4 % 2` gives 0.
- `3 / 2` gives 1, so `3 % 2` gives 1.
- `2 / 2` gives 1, so `2 % 2` gives 0.
- `1 / 2` gives 0, so `1 % 2` gives 1.
- `0 / 2` gives 0, so `0 % 2` gives 0.
- `7 / 3` gives 2, so `7 % 3` gives 1.
- `19 / 5` gives

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.
- 2 / 2 gives 1, so 2 % 2 gives 0.
- 1 / 2 gives 0, so 1 % 2 gives 1.
- 0 / 2 gives 0, so 0 % 2 gives 0.
- 7 / 3 gives 2, so 7 % 3 gives 1.
- 19 / 5 gives 3,

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.
- 2 / 2 gives 1, so 2 % 2 gives 0.
- 1 / 2 gives 0, so 1 % 2 gives 1.
- 0 / 2 gives 0, so 0 % 2 gives 0.
- 7 / 3 gives 2, so 7 % 3 gives 1.
- 19 / 5 gives 3, so 19 % 5 gives

Examples

- 4 / 2 gives 2, so 4 % 2 gives 0.
- 3 / 2 gives 1, so 3 % 2 gives 1.
- 2 / 2 gives 1, so 2 % 2 gives 0.
- 1 / 2 gives 0, so 1 % 2 gives 1.
- 0 / 2 gives 0, so 0 % 2 gives 0.
- 7 / 3 gives 2, so 7 % 3 gives 1.
- 19 / 5 gives 3, so 19 % 5 gives 4.

Modular Arithmetic

Assumption: ⟨rhs⟩ Greater than Zero

- `0 % ⟨rhs⟩` gives 0;
- `1 % ⟨rhs⟩` gives 1;
- …
- `(⟨rhs⟩ - 1) % ⟨rhs⟩` gives `⟨rhs⟩ - 1`;
- `⟨rhs⟩ % ⟨rhs⟩` gives 0;
- `(⟨rhs⟩ + 1) % ⟨rhs⟩` gives 1;
- ….

Modular/Clock Arithmetic

- `0 % 2` gives 0;
- `1 % 2` gives 1;
- `2 % 2` gives 0;
- `3 % 2` gives 1;
- `4 % 2` gives 0;
- ....

Application: Formatting Numbers

Java

```
/**
 * Output a number in the range 0..99 right formatted.
 * @param quantity The number.
 * @param nextString A string that's printed after the number.
 */
private static void format( final int quantity, final String nextString ) {
    System.out.print( (quantity / 10) );
    System.out.print( (quantity % 10) );
    System.out.print( nextString );
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Application: Displaying Time

Java

```
public class TimeFormatter {
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int HOURS_ON_CLOCK_FACE = 12;

    private static final String SEPARATOR = ":";
    private static final String EMPTY_STRING = "";

    public static void main( String[] args ) {
        final int time = 25 * 3600 + 35 * 60 + 7;

        final int secondsOnClock = time % SECONDS_PER_MINUTE;
        final int minutes = time / SECONDS_PER_MINUTE;
        final int minutesOnClock = minutes % MINUTES_PER_HOUR;
        final int hours = minutes / MINUTES_PER_HOUR;
        final int hoursOnClock = hours % HOURS_ON_CLOCK_FACE

        format( hoursOnClock, SEPARATOR );
        format( minutesOnClock, SEPARATOR );
        format( secondsOnClock, EMPTY_STRING );
    }
}
```

Evaluating Simple Expressions

- Simple expressions are easy to evaluate.

  $\langle\texttt{variable}\rangle_1\ \langle\texttt{binary arithmetic operator}\rangle\langle\texttt{variable}\rangle_2$,

- In general the order of evaluation matters, for example:

  Assignments: Sub-computations may carry out assignments.

Java

```java
int a = 2;
int b = a * (a = 1);
```

Side effects: Order also matters with other side-effects.

Associativity

left-to-right: Almost all operators are left associative:

$$v_1 \oplus v_2 \oplus \cdots \oplus v_n = (((v_1 \oplus v_2) \oplus \cdots) \oplus v_n\,.$$

Java

```
int answer = 840 / 10 / 2; // Assigns 42.
```

right-to-left: Only a few operators are right associative:

$$v_n \oplus \cdots \oplus v_2 \oplus v_1 = v_n \oplus (\cdots \oplus (v_2 \oplus v_1)).$$

Java

```
int result1, result2, result3;
result3 = result2 = result1 = 1;
// result3 = (result2 = (result1 = 1));
```

Arguments of Methods

Arguments of methods are also evaluated from left to right.

Java

```
private static int add( int first, int second ) {
    return first + second;
}

private void example( ) {
    int number = 0;
    int result = add( number = 1, number + 1 );
    System.out.println( result );
}
```

Arguments of Methods

Arguments of methods are also evaluated from left to right.

Java

```java
private static int add( int first, int second ) {
    return first + second;
}

private void example( ) {
    int number = 0;
    int result = add( number = 1, number + 1 );
    System.out.println( result );
}
```

Arguments of Methods

Arguments of methods are also evaluated from left to right.

Java

```
private static int add( int first, int second ) {
    return first + second;
}

private void example( ) {
    int number = 0;
    int result = add( number = 1, number + 1 );
    System.out.println( result );
}
```

Arguments of Methods

Arguments of methods are also evaluated from left to right.

Java

```java
private static int add( int first, int second ) {
    return first + second;
}

private void example( ) {
    int number = 0;
    int result = add( number = 1, number + 1 );
    System.out.println( result );
}
```

Arguments of Methods

Arguments of methods are also evaluated from left to right.

Java

```
private static int add( int first, int second ) {
    return first + second;
}

private void example( ) {
    int number = 0;
    int result = add( number = 1, number + 1 );
    System.out.println( result );
}
```

Arguments of Methods

Arguments of methods are also evaluated from left to right.

Java

```
private static int add( int first, int second ) {
    return first + second;
}

private void example( ) {
    int number = 0;
    int result = add( number = 1, number + 1 );
    System.out.println( result );
}
```

Arguments of Methods

Arguments of methods are also evaluated from left to right.

Java

```
private static int add( int first, int second ) {
    return first + second;
}

private void example( ) {
    int number = 0;
    int result = add( number = 1, number + 1 );
    System.out.println( result );
}
```

'Reasoning about expressions with sub-assignments and other side-effects is difficult. Avoid side-effects in expressions or else....'—Anonymous Java Lecturer.

Precedence

- In general Java expressions are evaluated from left to right.
- However, some operators should be applied before others.
- These operators are said to have a higher *precedence*.

Java

```
int three = 1 + 1 * 2; // Assigns 3 to three.
```

Override Operator Precedence

- It is always possible to override precedence with parentheses.

Java

```
int three =  1 + 1  * 2; // Assigns 3 to three.
int four  = (1 + 1) * 2; // Assigns 4 to four.
```

- Most programmers don't know exact operator precedences.
- Even if they do, they usually use parentheses for clarity:

Java

```
int result = 1 + ((2 * 3) / 4) + 5;
```

Mixed-Type Arithmetic

- We've seen that `Java` is strongly typed.
  - The type of all expressions must make "sense."
- Still, `Java` is pretty flexible.
  - For example, you can write '`1 + 2.3`.'
- So how does this work?

Widening

- Consider the expression ‘⟨`expr`⟩$_1$ `+` ⟨`expr`⟩$_2$,’ where
  - The type of ⟨`expr`⟩$_1$ is ⟨`type`⟩$_1$, and
  - The type of ⟨`expr`⟩$_2$ is ⟨`type`⟩$_2$.
- Let ⟨`type`⟩ be the type with larger range.
- The expression ‘`1 + 2.3`’ is evaluated using the type ⟨`type`⟩.
- However, ⟨`expr`⟩$_1$ and ⟨`expr`⟩$_2$ are first converted to ⟨`type`⟩.
- Next the resulting expressions are added.
- The result has the type ⟨`type`⟩.

Example

- For example, consider the expression ‘`1 + 2.3`.’
- The expression ‘`1`’ is an `int` literal.
- The expression ‘`2.3`’ is a `double` literal.
- The type `double` has the larger range.
- `Java` automatically converts the `int` to a `double`.
- The result of this *widening* conversion is `1.0`.
- Next the operator is applied.
- This results in `3.3`.

Primitive Type Widening

- When a primitive type value is widened, you cannot lose range.
- However, it may lose information because of rounding.
- Here rounding may occur with the following conversions:
  - `int` or `long` to `float`, and
  - `float` to a `double`,
- Still it is guaranteed that rounding is minimal.

Possible Widening Conversions

Source Type						Target Type	
	`byte`	`short`	`char`	`int`	`long`	`float`	`double`
`byte`		✓		✓	✓	✓	✓
`short`				✓	✓	✓	✓
`char`				✓	✓	✓	✓
`int`					✓	✓	✓
`long`						✓	✓
`float`							✓

Casting

- `Java` automatically widens `int`s to `double`.
- Converting from `double` to `int` is also possible.
  - In general conversions between numeric types are always possible.
  - It should be clear that you may get conversion errors.
- To convert `double source` to `int`, you write `(int)source`.
- This is called *casting* the `double source` to an `int`.
- Casting is also possible with other numeric types.

For Monday

- Study Chapter 2.
- Read Sections 3.1–3.2.
- Answer Review Questions R2.1–R2.5, R2.21, and R2.24.

Acknowledgements

- This lecture corresponds to [*Big Java, Early Objects*, 3.1–3.2].

Bibliography

Horstfmann, Cay S. *Big Java, Early Objects*. International Student Version. Wiley. ISBN: 978-1-118-31877-5.

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.