

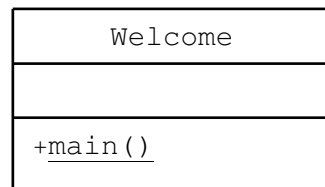
Anatomy of a Java Program

Starting Point

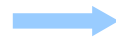
- A Java program is comprised of classes
 - *How many classes should I define in my program?*
 - *How to declare attributes and methods?*

. . . There will be many other questions after these ones are answered
- Let's analyze a simplest possible program that has only one class and this class has only one method `main()`

UML



*Even a simple
program must have
at least one class*



source file: Welcome.java

```
/* --- A simple Java program --- */
class Welcome
{
    public static void main(String[] args)
    {
        System.out.print("Welcome !");
    }
} // end of class Welcome
```

Class definition

Class declaration

Class body

```
class Welcome
{
}
}
```

To define a class you need to:

1. Use the `class` keyword (spelled with all lowercase letters)
2. Specify its name. A class name is its identifier.

By convention, class names

- can include only letters, digits, `_` and `$`. No spaces.
- begin with a capital letter. If several words are combined, each shall start with a capital letter too
- should be meaningful ~~`DpGh`, `Abc`, `R1`~~

`WelcomeToJava`, `StaffMember`, `StreamBuffer`

3. Provide a pair of curly braces `{ }` to specify the class body

Definition of a method main()

method declaration

method body

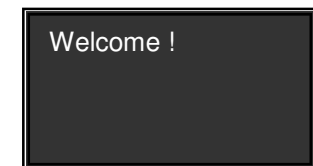
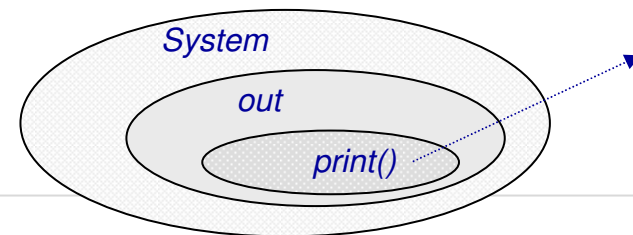
```
public static void main( String[] args )  
{  
  
}  
}
```

- Method `main()` is a starting point of every Java program. Program execution starts here. This method is called automatically by the JVM when you start a program.
- Keywords which must be used in the declaration:
 - `public` is an access specifier that indicates that `main()` is visible outside the class
 - `static` indicates that this method is placed in shared memory and becomes available right after the program start (even before objects are loaded in memory)
 - `void` indicates that this method doesn't return any information
- The method parameter `String[] args` will be explained later

Display a message

```
System.out.print("Welcome !");
```

- Instructs the JVM to display a message that is included in double quotes
- This is an **executable statement**. All executable statements must be terminated with the semicolon ;
- The statement executes a method `print()` that belongs to the object `out`
- The object `out` is a `static` object that belongs to the class `System`. The `out` is the standard output object that allows the JVM to display information in the command window
- The `System` is one of the Java API classes
- The dot (`.`) means "has"
System has **out** that has **print()**



Comments

- Even a well written program may be hard to follow if it implements complex data processing algorithms, or has a complex architecture
- It is a common programming practice to insert comments to improve readability of programs
- Comments in Java programs are indicated by special combination of characters
 1. `/*` these comments can span several lines `*/`

```
/*  A simple Java program
   UOW , SCIT 2017          */
```
 2. `//` these are one line comments

```
// calculation of the transform coefficient
```
 3. `/**` these comments are processed by javadoc program `*/`
- Comments are ignored by the compiler and have no affect on the program execution

Program 2: add two numbers

- The major purpose of programs is to process data
- Let's implement a simple program that calculates a sum of two numbers and displays the result

```
/* A simple program that calculates the sum */  
class SimpleAdder  
{  
    public static void main(String[] args)  
    {  
        // calculate and print the result  
        System.out.print("The sum = ");  
        System.out.println( 3.5 + 4.5 );  
    }  
}
```

an expression

*What should be
a name of the
file to store this
source code?*

- `println()` is another method that belongs to the object `out`
- `println()` displays the result and then moves to the next line
- these methods can display text and numbers

Program 3: calculate the average

- The previous example calculates the result and instantly displays it. This cannot be postponed as the program **does not store data**
- Let's implement another program that calculates an average of two numbers and displays the result
- This example introduces new fundamental elements: **variables**

DataProcessor.java

```
/* A simple program that calculates the average */  
class DataProcessor  
{  
    public static void main(String[] args)  
    {  
        double number1, number2, average; // declare variables  
        number1 = 3.2; // assign a value to number1  
        number2 = 6.8; // assign a value to number2  
        average = (number1 + number2)/2; // calculate average  
        // print the result  
        System.out.print("The average = ");  
        System.out.println( average ); // print a stored value  
    }  
} // end of class DataProcessor
```


Variables

- Even simple programs need to store data
- Computer programs store data in computer memory



How could I know which part of memory is free to store data and how data can be retrieved from memory when they are needed again?

- Java uses the concept of variables – **named** locations in memory. When you use variables you don't need to worry about program memory space addressing and management



*I study at ...latitude... longitude...
~~41°24'12.2"N 2°10'26.5"E~~
University of Wollongong*

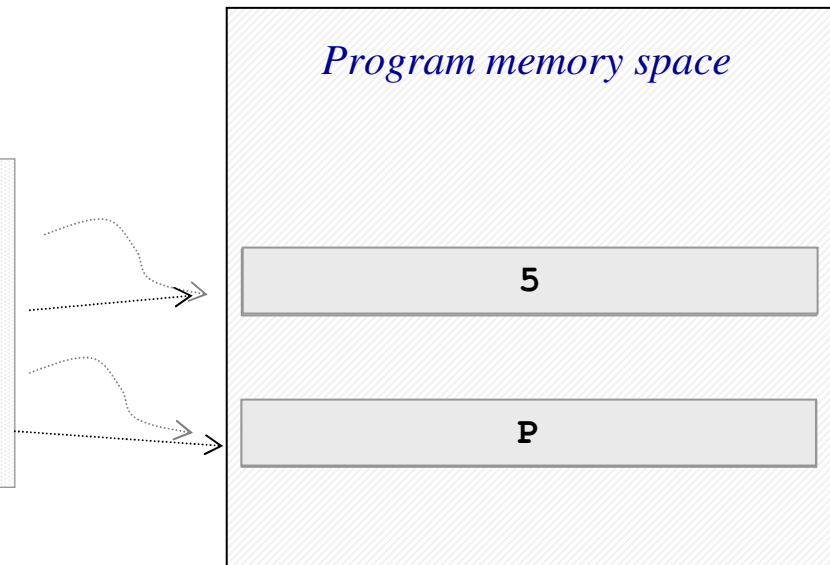
Named locations are simple to use

Variables

- You need to inform the compiler that your program needs a variable to store data - you need to **declare** a variable:
 - give it a name
 - specify what type of data you will be storing
- Based upon this declaration the compiler will:
 - find a free block of the memory space for this variable
 - reserve enough space to store the specified type of data

Example:

```
int number;    // declare a variable
number = 5;    // use the variable
char grade;    // declare a variable
grade = 'P';   // use the variable
```



Declaration of variables

- Declaration of a variable is a non-executable statement and therefore it must be terminated with `;` (as all Java statements)

Data type *Identifier (name)* `;`

```
int number;
```

- **Data type** is telling the compiler what type of data will be stored in a variable: a character, a whole number, a real number, ...
- **Identifier** can include only letters, digits, `_` and `$`. No spaces

Java naming convention for variables: lowercase for the first word and uppercase for the first character of subsequent words

Examples:

```
int numberOfDays;  
int itemsInStock, code, frameRate;  
char grade, inputCharacter;  
double totalPrice;
```

Java reserved words

Words reserved in the Java programming language that are forbidden for identifiers (class/method/variable names)

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	
continue	goto	package	synchronized	

All Java reserved words are lowercase

Declaration of variables

- A variable can be declared anywhere inside a method when you need it. You can place all declarations of variables at the beginning of methods and provide comments explaining what they are for
- If you declare variables outside of a method, these variables become fields (or data fields). In this case you may need to add their access specifier (public, private, etc)

```
class Example
{
    private double totalPrice; // total price of goods
    public calculatePrice( )
    {
        double currentPrice; // current price
        . . .
    }
}
```

a data field

a local variable

- If you give proper names, you may need less comments

Declaration of variables

- Declared data fields are implicitly initialised to 0. **Local variables are not**
- You can explicitly initialise local variables to specific values at the time of declaration. Alternatively, you can assign a value after the declaration.

```
int numberOfDays = 7;  
char grade = 'P';  
double initialTemperature = 100.0;
```

- Fixed values which you use in your source code are called **literals**
- Literals do not have any designated location in memory
- Literals can be specified using different notations, or numbering systems

```
double distance = 2.34e6;
```

scientific notation for 2.34×10^6

```
char code = '\n';
```

variable

literal

Literals of invisible characters are specified through escape sequences

`\n` – *new line*, direct output to the next line of the command window

`\t` – *horizontal tab*

Arithmetic operators and expressions

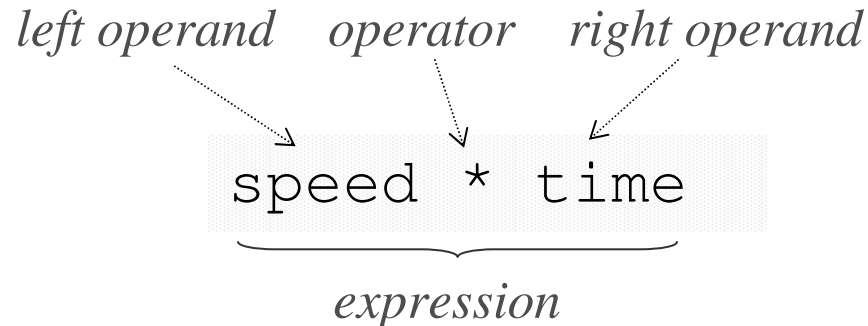
- Once a variable has been declared and initialised, it can be used for data storage and processing
- Java defines a set of **arithmetic operators**

+ addition
- subtraction
* multiplication
/ division
. . . .

left operand *operator* *right operand*

speed * time

expression



- Operators which require two operands are called **binary operators**
- A sequence of **operands** and **operators** that produces a value is called an **expression**

Examples of expressions:

```
4.5 * newValue - oldValue  
(number1 + number2) / 2.0  
3.14159 * radius * radius
```

Assignment operator

- Besides arithmetic operators, there are other operators
- An **assignment operator** assigns value of its right operand to its left operand

```
lvalue = rvalue
```

- `lvalue` is a program component that must have an designated location in memory (a variable)
- `rvalue` does not have a designated location in memory and can be a literal, a constant, or a value produced by an expression

Example:

```
int number = 25;           // OK
number = number + 7;       // OK
25 = number;               // ERROR (literal 25 is not an
                             lvalue)
(number+1) = 25;           // ERROR (expression is not an
                             lvalue)
```


Symbolic constants

- Although it looks simple to use literals in expressions, sometimes it may cause inconvenience

Example: You implement a program that uses conversion of miles into kilometres. The conversion is carried out according to a simple formula

*kilometers = 1.609 * miles*

You implement this formula using 1.609 literal in your program. It works.

Design requirements are changing: you need to use nautical miles instead

*kilometers = 1.852 * miles*

The change looks simple but... it needs to be done in 35 methods of 12 classes

- It is more practical to define a symbolic constant equal to a literal if there is a chance that the literal may be changed in future

```
final double COEFF = 1.609;
```

this keyword indicates a constant

*Name of the constant
Java notation - uppercase*

*The actual value of the
symbolic constant COEFF*

Symbolic constants

- Define a symbolic constant when a fixed value is needed
- Use it everywhere in your program instead of literals
- If the actual value needs to be changed, the literal needs to be changed only in the constant declaration statement

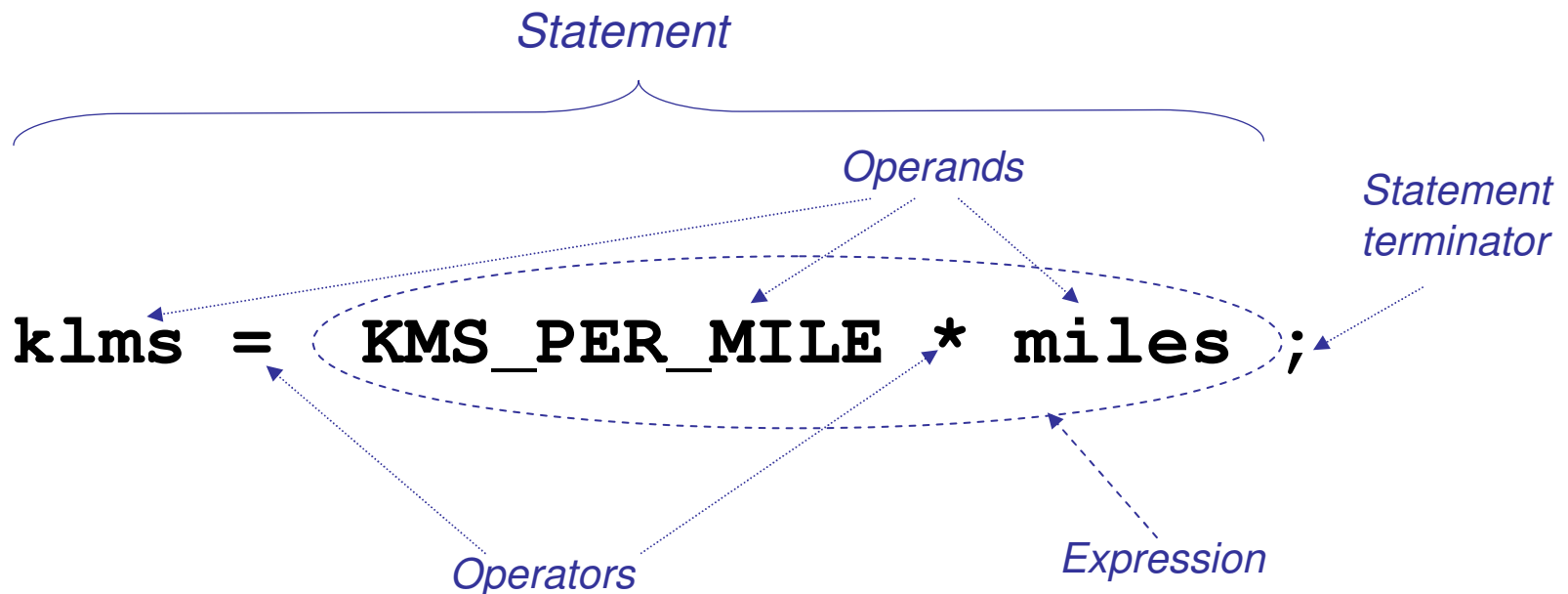
```
public static void main(String[] args)
{
    . . .
    final double COEFF = 1.609; // conversion coefficient for English mile
    final double PI = 3.1416;   // a more accurate value may be needed

    . . .
    distInKilom = COEFF * distInMiles;

    . . .
    area1 = PI * radius1 * radius1;
    . . .
    area2 = PI * radius2 * radius2;
}
```

Quiz

Identify all elements: operators, operands, expressions, statements, rvalues, lvalues



Quiz

This code cannot be compiled. What is wrong?

```
class Example
{
    public static void main(String[] args)
    {
        double num1= 0.5, num2, num3, result; // declare variables
        num2 = 4.0; // assign a value to num2

        result = (num1 + num2) * num3;

        System.out.println( result ); // print the result
    }
}
```

Math functions

- Java defines only a limited set of arithmetic operators (there are some others to be discussed later) that may not be enough to implement complex calculations
- Java is an extensible language. It has a library *Math* of methods for performing the most common numeric operations

```
phase = Math.sin( 2.0*Math.PI*f*t );
```

library name

method name

*Constant π
defined in Math*

- *Math* is implemented as a class with all fields and methods declared as `static`

Math functions

public class Math

double abs(double a)	<i>absolute value of a</i>
double max(double a, double b)	<i>maximum of a and b</i>
double min(double a, double b)	<i>minimum of a and b</i>
double sin(double theta)	<i>sine of theta</i>
double cos(double theta)	<i>cosine of theta</i>
double tan(double theta)	<i>tangent of theta</i>
double toRadians(double degrees)	<i>convert angle from degrees to radians</i>
double toDegrees(double radians)	<i>convert angle from radians to degrees</i>
double exp(double a)	<i>exponential (e^a)</i>
double log(double a)	<i>natural log ($\log_e a$, or $\ln a$)</i>
double pow(double a, double b)	<i>raise a to the bth power (a^b)</i>
long round(double a)	<i>round a to the nearest integer</i>
double random()	<i>random number in [0, 1]</i>
double sqrt(double a)	<i>square root of a</i>
double E	<i>value of e (constant)</i>
double PI	<i>value of π (constant)</i>

The import statement

- It may be a bit inconvenient to type `Math` every time you use a method or a constant defined in this class
- If you simply omit the class name, the compiler will not be able to find these methods and constants as they are defined inside the class
- Java provides an import statement that notifies the compiler where to find the required methods, constant, or classes

```
/* An example that shows how to use the import statement */
import static java.lang.Math.* // import all members of Math class

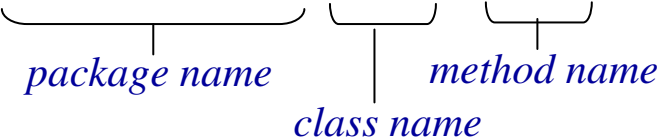
class SignalProcessor
{
    . . .
    public double getPhase( double time, double f )
    {
        . . .
        phase = sin( 2.0*PI*f*time );
        . . .
    }
}
```

You don't need to specify `Math` explicitly as it is specified implicitly through the import statement

Java packages

- The Java designers have proposed a flexible network based environment where classes can be loaded if their directories are accessible (locally, or via networks)
- There are currently more than 4000 classes in Java API and all visible identifiers must be unique (which is hard to achieve)
- To avoid collision of names, Java classes are grouped together into separated namespaces which are called **packages**
- Actually, to use any Java component you need to specify its full name including its package

Example: `ph = java.lang.Math.cos(0.5);`



- Instead of writing full names it is more convenient to use `import`
- The package `java.lang` is imported implicitly so you need not write `import java.lang.*;` in your programs

Quiz

Does import statement

```
import java.util.Random;
```

moves class Random into your application?

What is the difference between

```
import java.util.Random;
```

and

```
import java.util.*;
```

A simple application scenario

- Programming is a problem solving activity that involves understanding of the application area and analysis of the problem

Write a program that converts distance measured in miles into kilometres

1. Analyze the problem (input data, output data, data processing algorithm)

Input: distance in miles

Output: distance in kilometres

Algorithm: `kilometres = 1.609 * miles`

2. Use UML to describe a class
3. Implement the program

Converter
<u>-distInMiles:</u> <u>-distInKilom:</u> <u>-COEFFICIENT</u> = 1.609 {read only}
+ <u>main()</u>

Java implementation

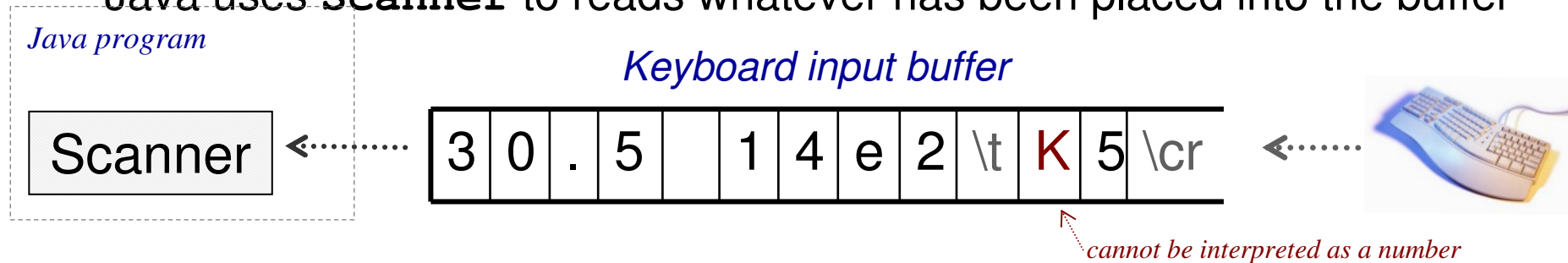
Converter.java

```
/* A program that converts miles into kilometres
   input: distance in miles -> output: distance in kilometres */
import java.util.Scanner; // import Scanner class
class Converter
{ // --- fields ---
    private static double distInMiles;
    private static double distInKilom;
    private static final double COEFF = 1.609;
    // --- definition of the method main() ---
    public static void main(String[] args)
    {
        // Create a scanner that can read numbers, or words
        Scanner input = new Scanner(System.in);
        System.out.print("Enter distance in miles:");
        distInMiles = input.nextDouble(); // input a number typed by a user
        // calculate the distance
        distInKilom = COEFF * distInMiles;
        // output the result
        System.out.println("Distance = " + distInKilom + " km");
    }
} // end of class Converter
```

Distance = 160.9 km

Keyboard input

- **Scanner** methods `nextInt()`, `nextFloat()` and `nextDouble()` do not read keyboard input directly (Java cannot have direct access to hardware)
- Everything what you type goes into the keyboard input buffer first (processed by the operating system modules)
- Java uses **Scanner** to reads whatever has been placed into the buffer



- **Scanner** methods `nextInt()`, `nextFloat()` and `nextDouble()` read all digits until a whitespace (*tab, space, newline*) is found
- If digits cannot be interpreted as a number, this leads to a run-time error
- **Scanner** method `nextLine()` reads all characters until *cr* (*Enter*)

Java implementation v2

Declaration of fields makes sense when a class has several methods which can access the fields. If there is only one method in a class, all data can be stored in local variables declared in this method

```
import static java.lang.System.*; // import all members of System
import java.util.Scanner;         // import Scanner class
class Converter
{
    public static void main(String[] args)
    {
        double distInMiles, distInKilom; // local variables
        final double COEFF = 1.609;      // a local constant

        Scanner input = new Scanner( in );
        out.print("Enter distance in miles:");
        distInMiles = input.nextDouble(); // input the typed number

        distInKilom = COEFF * distInMiles;
        // formatted output
        out.printf("%.1f miles = %.1f km \n", distInMiles, distInKilom);
    }
}
```

100.0 miles = 160.9 km

Formatted output

- Methods `print()` and `println()` are simple, but can provide only very basic output
- Method `printf()` provides a flexible way to output numbers, characters and text in a specified format using only one statement

Examples: Assuming the following variables

```
int date=25, month=2, year=2017;
```

- Print date as **25**

```
printf("%d", date);
```

- Print date as **Today is 25th** and direct output to the next line

```
printf("Today is %dth \n", date);
```

- Print date, month and year as **Current date: 25/02/2017**

```
printf("Current date: %02d/%02d/%d", date, month, year);
```

printf(...) method

format string print list

```
printf("%f miles equal to %f kilometres \n", miles, kms);
```

text *Formatting parameters* *Escape sequence*
\n – new line

The diagram illustrates the components of the `printf` function call. A bracket labeled 'format string' spans the string `"%f miles equal to %f kilometres \n"`. Another bracket labeled 'print list' spans the list `miles, kms`. Below the code, three labels with arrows point to specific parts: 'text' points to the first `%f`, 'Formatting parameters' points to the second `%f`, and 'Escape sequence' points to the `\n`. A note below the last label states '`\n` – new line'.

Example: If `miles = 10.0` and `kms = 16.09`

10.000000 miles equal to 16.090000 kilometres

printf (...) method

- Formatting parameters

% flag width .prec type

Grayed parameters are optional.

If they are used, they must appear in this order

type	Interpretation
d	Display as an integer
f	Display as a floating point number (with six decimal places by default)
e	Display a floating-point number in exponential format
c	Display a single character

width & .prec	Meaning
<i>number</i>	This number of positions will be reserved for the output
<i>.number</i>	This number of decimal places will be used for f formats

printf(...) method

```
double weight = 12.978;  
double temp = -97.4583;
```

```
printf("Doubles:\n");  
printf("%f %e\n", weight, weight);  
printf("%.2f %.2e\n", weight, weight);  
printf("%.0f %.0e\n", weight, weight);  
printf("%9.2f %9.2f\n", weight, temp);
```

```
Doubles:  
12.978000 1.297800e+001  
12.98 1.30e+001  
13 1e+001  
12.98 -97.46
```

9 positions reserved *9 positions reserved*

Binary Numeral System

Source code

```
byte a = 5;
```



Program memory

```
11111011
```

- All numbers are stored in computer memory in binary format
- Conversion between decimal and binary numbering systems is carried out by the compiler
- Without understanding of the binary numeral system you will not be able to
 - use correctly some of the operators supported by Java
 - explain some of the calculation errors
- Typical application areas
 - network communication protocols (error detection, flow control, etc)
 - data encryption and data compression
 - computer graphics

Binary vs. Decimal

- The decimal system uses ten symbols to represent numbers

Symbol set: 0 1 2 3 4 5 6 7 8 9

- The binary system uses two symbols to represent numbers

Symbol set: 0 1

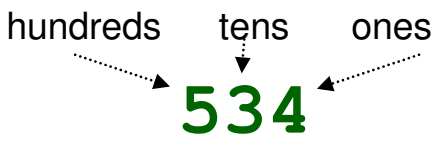
Example:

one	1	1
two	2	10
three	3	11
four	4	100

Binary vs. Decimal

- Apart from symbol sets, there is no conceptual difference between decimal and binary systems
- Both systems are positional

Example:

five hundred thirty four: 

$$534 = 5 * 10^2 + 3 * 10^1 + 4 * 10^0$$

10 is a base

A weight of each symbol depends on its position

Hundred in old-English "*hund*" means "*tens*", "*rad*" means "*number*"

Binary vs. Decimal

- Binary number

Example:

eights fours twos ones
 ↓ ↓ ↓ ↓
 1011

$$\mathbf{1011} = \mathbf{1} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{1} * 2^0$$

2 is a base

A weight of each symbol also depends on its position

The binary system and binary arithmetic was invented by Gottfried Leibniz and published in 1701 in Paris.

The first computer utilising the binary system was built in 1946 at the University of Pennsylvania.

Binary to Decimal conversion

- Binary number

$$101011 = 1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$$

2^0	=	1
2^1	=	2
2^2	=	4
2^3	=	8
2^4	=	16
2^5	=	32
2^6	=	64
2^7	=	128

$$101011 = 1*32 + 0 + 1*8 + 0 + 1*2 + 1*1 = 43$$

Quiz

- Convert to decimal

$$0000 = 0*8 + 0*4 + 0*2 + 0 = 0$$

$$1000 = 1*8 + 0*4 + 0*2 + 0 = 8$$

$$1111 = 1*8 + 1*4 + 1*2 + 1 = 15$$

2^0	= 1
2^1	= 2
2^2	= 4
2^3	= 8
2^4	= 16
2^5	= 32
2^6	= 64
2^7	= 128

Decimal to Binary conversion

- Conversion from binary to decimal:
repetitive multiplication and addition
- Conversion from decimal to binary:
repetitive division by 2 and subtraction

Example: Convert 23 to binary

23	/	2	=	11	and the remainder =	1	
11	/	2	=	5	and the remainder =	1	
5	/	2	=	2	and the remainder =	1	
2	/	2	=	1	and the remainder =	0	
1	/	2	=	0	and the remainder =	1	

STOP

23 = 10111 ← *Result*

Quiz

► 1. Convert to binary: 5

The result

$$\begin{array}{lcl} 5 / 2 = 2 & \text{and the remainder} = & 1 \\ 2 / 2 = 1 & \text{and the remainder} = & 0 \\ 1 / 2 = 0 & \text{and the remainder} = & 1 \end{array} \left. \vphantom{\begin{array}{lcl} 5 / 2 = 2 \\ 2 / 2 = 1 \\ 1 / 2 = 0 \end{array}} \right\}$$

STOP

1 0 1

► 2. Convert to binary: 9

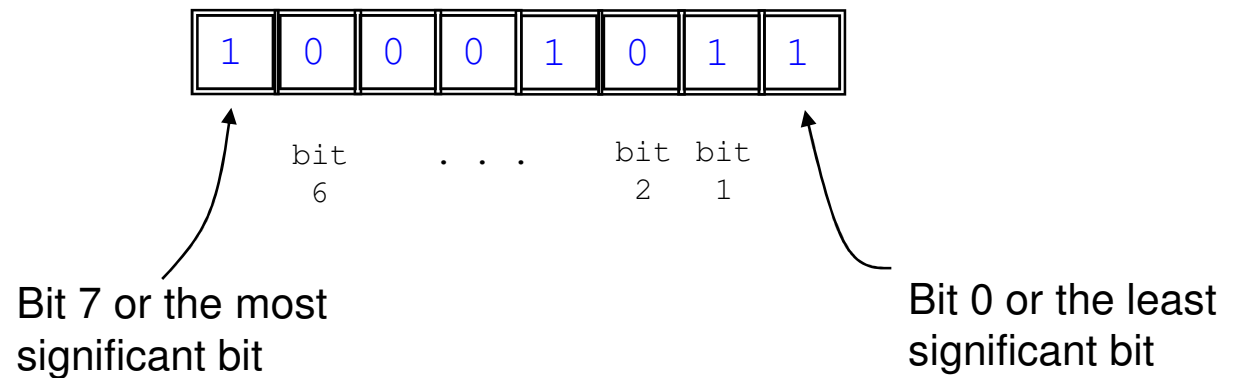
► The result is 1001

Terminology

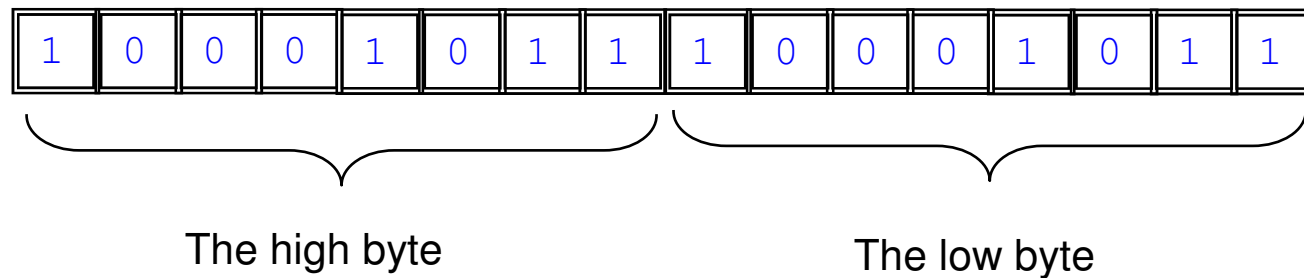
- A bit:



- A byte:



- A word:



Binary Addition

- Addition is carried out on individual bits taking into account carry bits

Basic rules:

$$0 + 0 = 0 \qquad 1 + 0 = 1 \qquad 1 + 1 = 10$$

Example:

Diagram illustrating binary addition:

Left side (Binary):

```

    carry bit 1
      1 0 1 0
    + 0 0 1 1
    -----
      1 1 0 1
  
```

Right side (Decimal):

```

    10
  +  3
  ----
   13
  
```

Signed Binary Numbers

- All Java variables are signed (they can store positive and negative values)
- Signed binary numbers are represented in two's complement format

unsigned format: 5 = 101

two's complement: 5 = 0101

two's complement: -5 = 1011

} *an extra bit is required to represent a sign*

- To change a negative unsigned binary number to its two's complement format:
 1. Write down its binary representation ignoring a sign: 101
 2. Attach an extra zero to the left: 0101
 3. Invert the number(change 0 to 1 and 1 to 0): 1010
 4. Add 1 to the number: 1011

Storing binary numbers

a byte – stores numbers of the range [-128 to 127]
in two's complement format

0	0	0	0	0	0	0	0	->	0
0	0	0	0	0	0	0	1	->	1
0	1	1	1	1	1	1	1	->	127
1	0	0	0	0	0	0	0	->	-128
1	1	1	1	1	1	1	1	->	-1

Suggested reading

Java: How to Program (Early Objects), 10th Edition

- Chapter 2: Introduction to Java applications