

Introduction to Programming

“ Programming is part practical, part theory




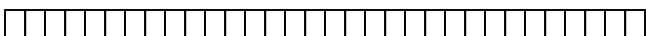

- If you are just practical, you produce non-scalable un-maintainable hacks
- If you are just theoretical, you produce toys ”

- Bjarne Stroustrup

Computer Language

- Digital devices have two stable states, which are referred to as zero and one, by convention.
- The binary number system has two digits, 0 and 1. A single digit (0 or 1) is called a *bit*, short for ***binary digit***. A byte is made up of 8 bits.
- Binary Language: Data and instructions (numbers, characters, strings, etc.) are encoded as binary numbers - a series of bits (one or more bytes made up of zeros and ones)

Information storage

- 1 bit 
 - 8 bits 
 - 16 bits 
 - 32 bits 
 - 64 bits 
- bit (1 or 0)
 - byte (octet) (2^8)
 - word (2^{16})
 - double (2^{32})
 - long double (2^{64})

Number Systems

- Decimal
 - Base 10, ten digits (0-9)
 - The position (place) values are integral powers of 10: 10^0 (ones), 10^1 (tens), 10^2 (hundreds), 10^3 (thousands)...
 - n decimal digits - 10^n unique values
- Binary
 - Base 2, two digits (0-1)
 - The position (place) values are integral powers of 2: 2^0 (1), 2^1 (2), 2^2 (4), 2^3 (8), 2^4 (16), 2^5 (32), 2^6 (64)...
 - n binary digits - 2^n unique values

Information coding

- Binary
 - 0 or 1
- Octal
 - 0-7
- Hexadecimal
 - 0-9, A-F
- Decimal
 - 0-9

- How to count

128	64	32	16	8	4	2	1
1	0	1	1	0	1	0	1

- $128+32+16+4+1=$
 - 181 (decimal)
 - 265 (octal)
 - B5 (hexadecimal)
- Signed *vs* unsigned
 - 0 to 255
 - -127 to +127

Computer Language (cont.)

- Encoding and decoding of data into binary is performed automatically by the system based on the encoding scheme
- Encoding schemes
 - Numeric Data: Encoded as binary numbers
 - Non-Numeric Data: Encoded as binary numbers (sequences) using representative code
 - ASCII – 1 byte per character
 - Unicode – 2 bytes per character

ASCII Table

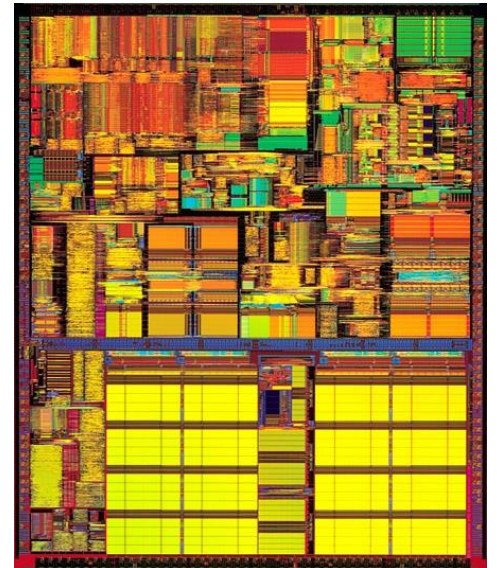
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	so	d;e	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	“	#	\$	%	&	‘
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

Programming Languages

- Computers can not use human languages, and programming in the binary language of computers is a very difficult, tedious process.
- Therefore, most programs are written using a programming language and are converted to the binary language used by the computer.
- Three major categories of programming languages:
 - Machine Language
 - Assembly Language
 - High level Language

Need for High Level Languages

- Computers are fast
 - Pentium 4 chip from 2001 can perform approximately 1,700,000,000 computations per second
 - made up of 42,000,000 transistors (a switch that is on or off)
- Computers are dumb
 - They can only carry out a very limited set of instructions
 - on the order of 100 or so depending on the computer's processor
 - machine language instructions, aka instruction set architecture (ISA)
 - Add, Branch, Jump, Get Data, Get Instruction, Store



Machine Language

- Natural language of a particular computer
- Primitive instructions built into every computer
- The instructions are in the form of binary code
- Any other type of language must be translated down to this level

Machine Code

- John von Neumann - co-author of paper in 1946 with Arthur W. Burks and Hermann H. Goldstine,
 - "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument"
- One of the key points
 - program commands and data stored as sequences of bits in the computer's memory

- A program:

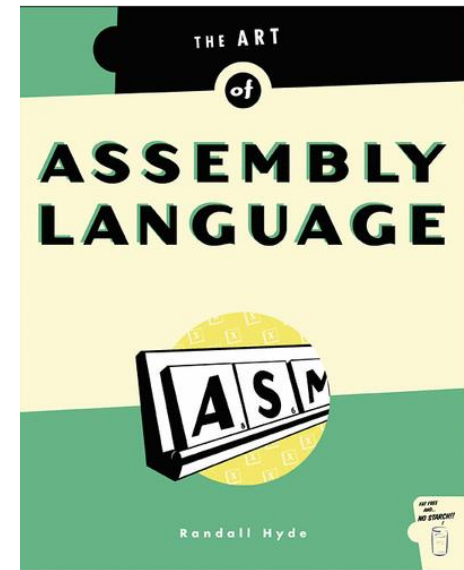
```
1110001100000000
0101011011100000
0110100001000000
0000100000001000
0001011011000100
0001001001100001
0110100001000000
```



Assembly Language

- Programming with Strings of bits (1s or 0s) is not the easiest thing in the world.
- Assembly language
 - mnemonics for machine language instructions

.ORIG	x3001
LD	R1, x3100
AND	R3, R3 #0
LD	R4, R1
BRn	x3008
ADD	R3, R3, R4
ADD	R1, R1, #1
LD	R4, R1
BRnzp	x3003



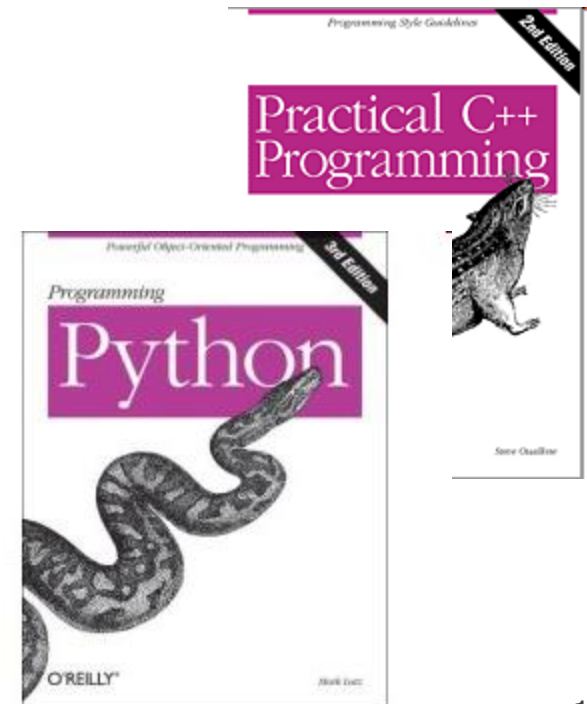
Assembly Languages

- English-like Abbreviations used for operations (Load R1, R8)
- Assembly languages were developed to make programming easier
- The computer cannot understand assembly language -- a program called assembler is used to convert assembly language programs into machine code

High Level Languages

- Assembly language is still not so easy, and needs lots of commands to accomplish things
- High Level Computer Languages provide the ability to accomplish a lot with fewer commands than machine or assembly language, in a way that is hopefully easier to understand

```
int sum;  
int count = 0;  
int done = -1;  
while( list[count] != -1 )  
    sum += list[count];
```



High Level Languages

- English-like, and easy to learn and program
- Common mathematical notation
 - $\text{Total Cost} = \text{Price} + \text{Tax};$
 - $\text{area} = 5 * 5 * 3.1415;$
- Java, C, C++, FORTRAN, VISUAL BASIC, PASCAL

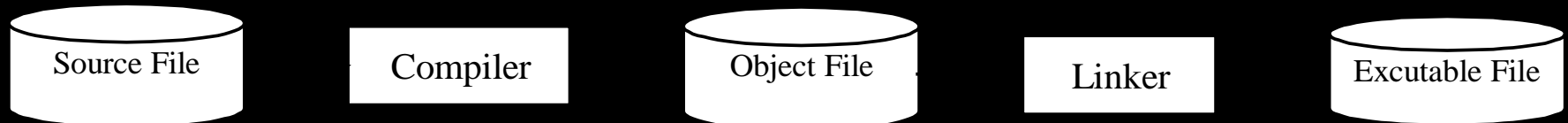
High Level Languages

- There are hundreds of high level computer languages. Java, C++, C, Basic, Fortran, Cobol, Lisp, Perl, Prolog, Eiffel, Python
- The capabilities of the languages vary widely, but they all need a way to do
 - declarative statements
 - conditional statements
 - iterative or repetitive statements
- A compiler is a program that converts commands in high level languages to machine language instructions

Compiling Source Code

- A program written in a high-level language is called a *source program* (or *source code*). Since a computer cannot understand a source program,

a program called *compiler* is used to translate the source program into a machine language program called an *object program*.
- The object program is often then linked with other supporting library code before it can be executed on the machine.



Compiling and Running

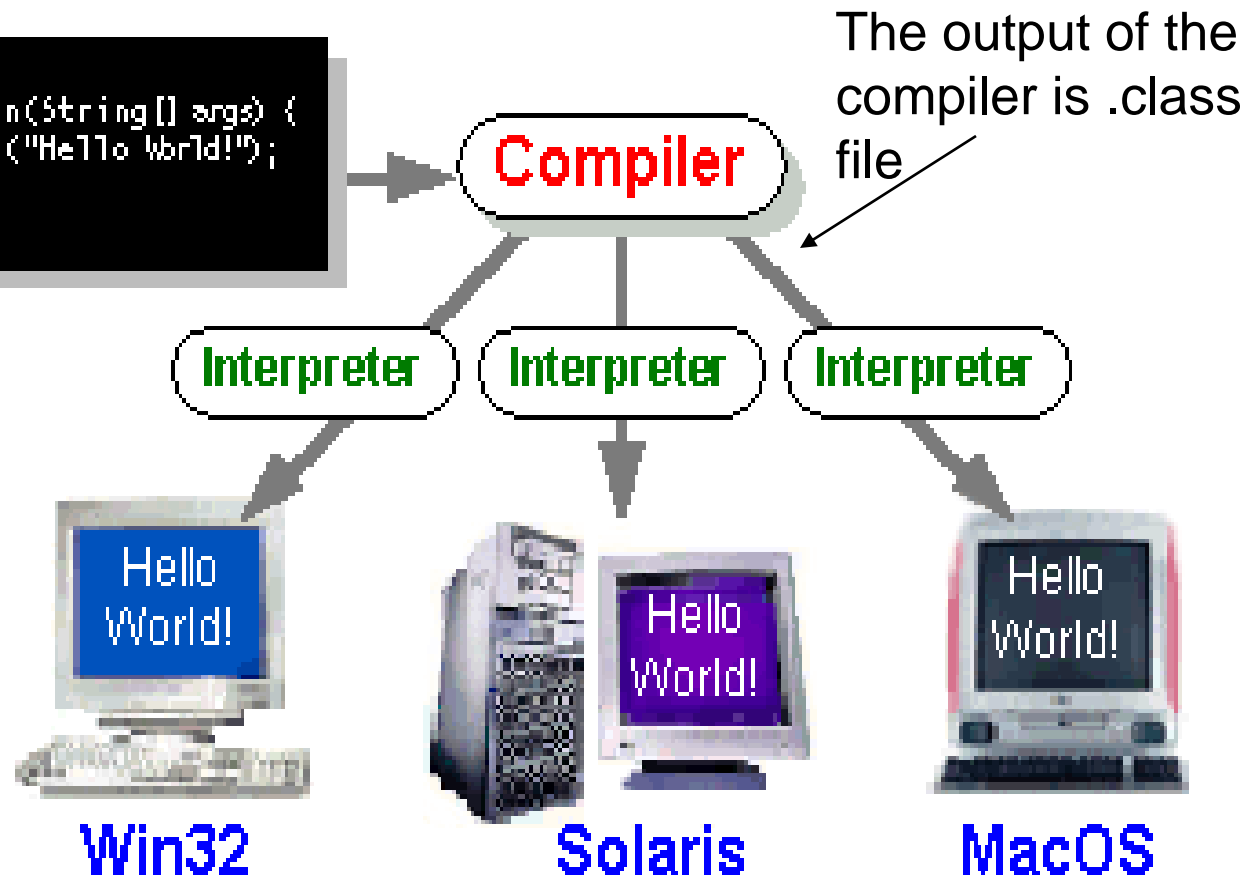
- **Compiler:** a program that converts a program in one language to another language
 - compile from C++ to machine code
 - compile Java to *bytecode*
- **Bytecode:** a language for an imaginary cpu
- **Interpreter:** converts one instruction or line of code from one language to another and then executes that instruction
 - When java programs are run, the bytecode produced by the compiler is fed to an interpreter that converts it to machine code for a particular CPU

Java Interpreter

Java Program

```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

HelloWorldApp.java



The Interpreters are sometimes referred to as the Java Virtual Machines

Compiling Java Source Code

- You can port a source program to any machine with appropriate compilers. The source program must be recompiled, however, because the object program can only run on a specific machine.
- Nowadays computers are networked to work together. Java was designed to run object programs on any platform. With Java, you write the program once, and compile the source program into a special type of object code, known as *bytecode*.
- The bytecode can then run on any computer with a Java Virtual Machine. Java Virtual Machine is a software that interprets Java bytecode.

Source code -> Object code

- Compiler+linker
 - Fortran, C, Pascal, C++...
 - Interpreter
 - Basic, Perl...
 - Intermediate
 - Java
- Compiler+linker
 - ♦ Fast to execute, but slow to debug
 - Interpreter
 - ♦ Slow to execute, but fast to debug (no need to recompile)
 - Intermediate
 - ♦ Slow...

What is programming?

- **program:** A set of instructions to be carried out by a computer.
(an algorithm written in a programming language)
- **program execution:** The act of carrying out the instructions contained in a program.
- **programming language:** A systematic set of rules used to describe computations in a format that is editable by humans.



What is programming?

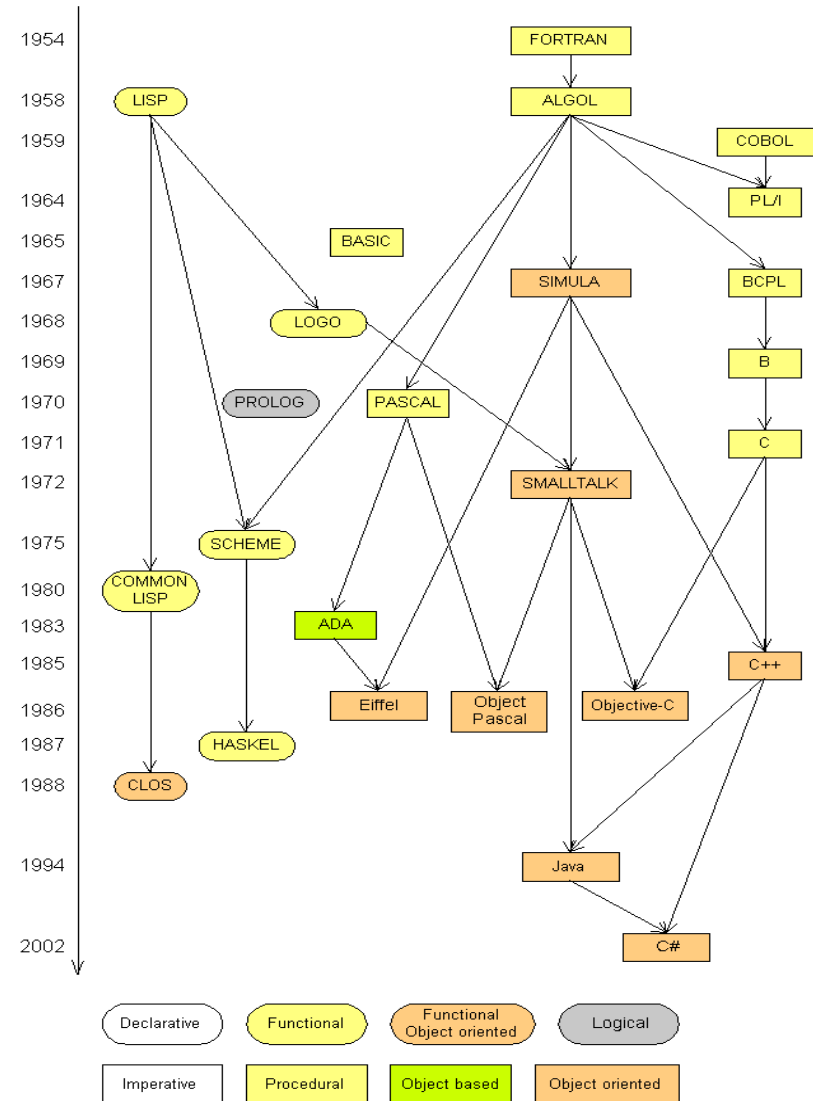
- **programming** – the creation of an ordered set of instructions to solve a problem with a computer.
- Only about 100 instructions that the computer understands - Different programs will just use these instructions in different orders and combinations.
- The most valuable part of learning to program is learning how to think about arranging the sequence of instructions to solve the problem or carry out the task

Programming Fundamentals: Putting the Instructions Together

- Sequential Processing
 - A List of Instructions
- Conditional Execution
 - Ifs
- Repetition
 - Looping / Repeating
- Stepwise Refinement / Top-Down Design
 - Breaking Things into Smaller Pieces
- Calling Methods / Functions / Procedures / Subroutines
 - Calling a segment of code located elsewhere
 - Reuse of previously coded code segment

Programming languages

- Some influential ones:
 - FORTRAN
 - science / engineering
 - COBOL
 - business data
 - LISP
 - logic and AI
 - BASIC
 - a simple language



Some modern languages

- *procedural languages*: programs are a series of commands
 - **Pascal** (1970): designed for education
 - **C** (1972): low-level operating systems and device drivers
- *functional programming*: functions map inputs to outputs
 - **Lisp** (1958) / **Scheme** (1975), **ML** (1973), **Haskell** (1990)
- *object-oriented languages*: programs use interacting "objects"
 - **Smalltalk** (1980): first major object-oriented language
 - **C++** (1985): "object-oriented" improvements to C
 - successful in industry; used to build major OSes such as Windows
 - **Java** (1995): designed for embedded systems, web apps/servers
 - Runs on many platforms (Windows, Mac, Linux, cell phones...)

Methods of Programming

- Procedural
 - Defining set of steps to transform inputs into outputs
 - Translating steps into code
 - Constructed as a set of procedures
 - Each procedure is a set of instructions
- Object-Oriented
 - Defining/utilizing objects to represent real-world entities that work together to solve problem
 - Basic O-O Programming Components
 - Class
 - Object/Instance
 - Properties
 - Methods

Object Oriented Programming

- Class

- Specifies the definition of a particular kind of object
 - Its Characteristics : **Properties** (or Attributes)
 - Its Behaviors: **Methods**
- Used as blueprint / template to create objects of that type

- Object/Instance

- A specific instance of a class – an object created using the Class Definition
- All specific instances of the same class share the same definition
 - Same Properties – can be modified
 - Same Methods – can be modified

Class and Object Example

Class (aLL DOGS)

- All Instances of Class Dog Have
 - Properties
 - Name
 - Breed
 - Weight
 - Color
 - Methods
 - Walk
 - Bark
 - Jump

Object (one dog)

- A particular Instance Could Have
 - Properties
 - Name -Spot
 - Breed - Mutt
 - Weight - 10 pounds
 - Color - Black
 - Methods
 - Walk
 - Bark
 - Jump

(these can also be modified to fit a particular dog)

Basic Java programs with `println` statements

A Java program

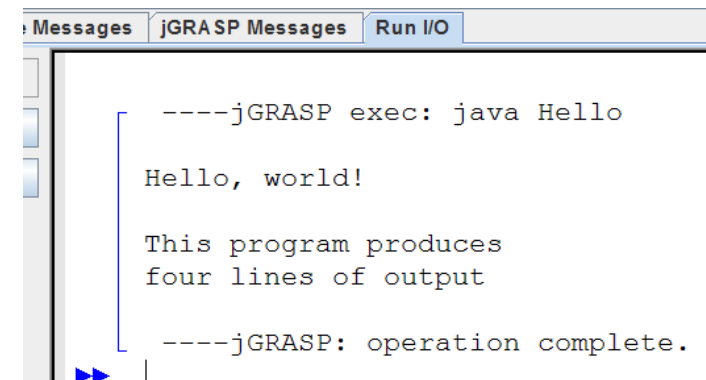
```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
        System.out.println();  
        System.out.println("This program produces");  
        System.out.println("four lines of output");  
    }  
}
```

- **Its output:**

Hello, world!

This program produces
four lines of output

- **console:** Text box into which the program's output is printed.



Compile/run a program

1. Write it.

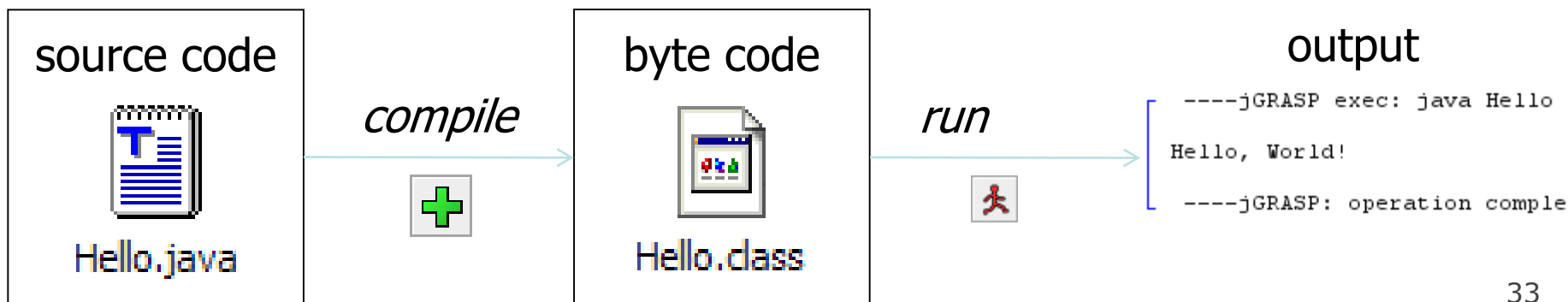
- **code** or **source code**: The set of instructions in a program.

2. Compile it.

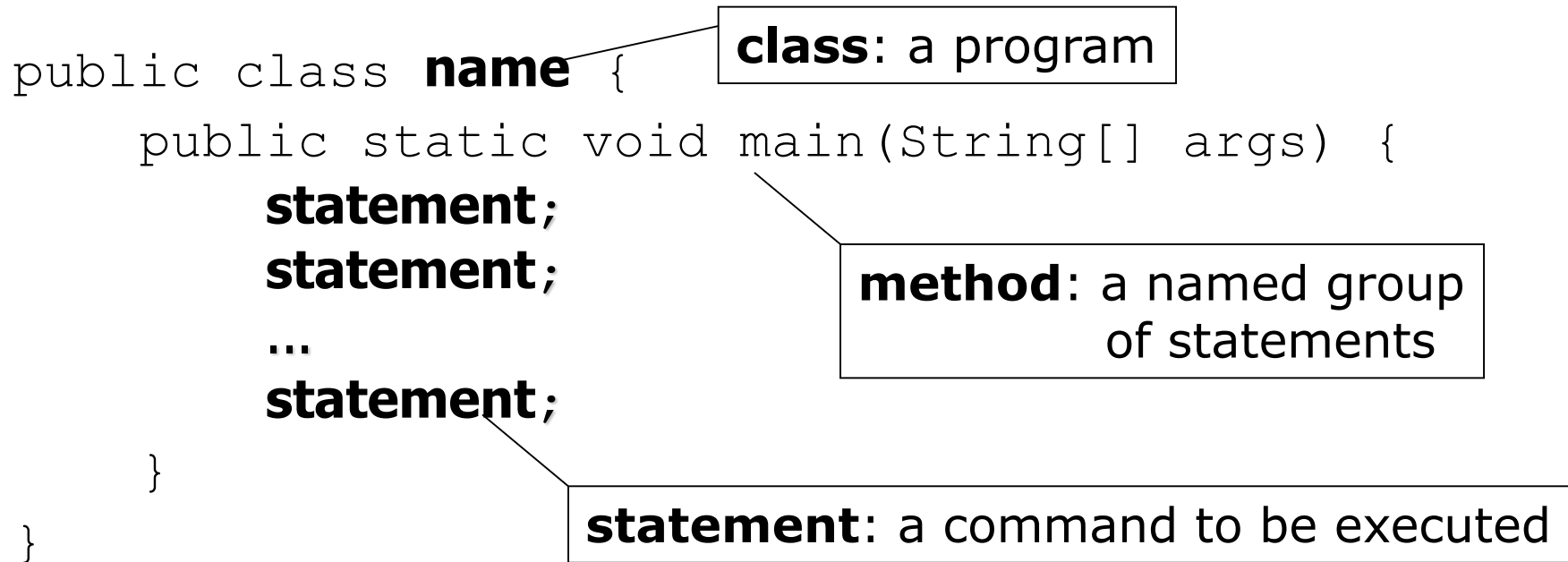
- **compile**: Translate a program from one language to another.
- **byte code**: The Java compiler converts your code into a format named *byte code* that runs on many computer types.

3. Run (execute) it.

- **output**: The messages printed to the user by a program.



Structure of a Java program



- Every executable Java program consists of a **class**,
 - that contains a **method** named `main`,
 - that contains the **statements** (commands) to be executed.

System.out.println

- A statement that prints a line of output on the console.
 - pronounced "print-linn"
 - sometimes called a "println statement" for short
- Two ways to use `System.out.println` :
 - `System.out.println("text") ;`
Prints the given message as output.
 - `System.out.println() ;`
Prints a blank line of output.

Names and identifiers

- You must give your program a name.

```
public class GangstaRap {
```

- Naming convention: capitalize each word (e.g. `MyClassName`)
- Your program's file must match exactly (`GangstaRap.java`)
 - includes capitalization (Java is "case-sensitive")

- **identifier**: A name given to an item in your program.

- must start with a letter or `_` or `$`
- subsequent characters can be any of those or a number

• **legal:** `_myName` `TheCure` `ANSWER_IS_42` `$bling$`

• **illegal:** `me+u` `49ers` `side-swipe` `Ph.D's`

Keywords

- **keyword:** An identifier that you cannot use because it already has a reserved meaning in Java.

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	

Syntax

- **syntax:** The set of legal structures and commands that can be used in a particular language.
 - Every basic Java statement ends with a semicolon ;
 - The contents of a class or method occur between { and }
- **syntax error (compiler error):** A problem in the structure of a program that causes the compiler to fail.
 - Missing semicolon
 - Too many or too few { } braces
 - Illegal identifier for class name
 - Class and file names do not match
 - ...

Syntax error example

```
1 public class Hello {  
2     poublic static void main(String[] args) {  
3         System.owt.println("Hello, world!")_  
4     }  
5 }
```

- Compiler output:

```
Hello.java:2: <identifier> expected  
    poublic static void main(String[] args) {  
      ^
```

```
Hello.java:3: ';' expected  
    }  
    ^
```

```
2 errors
```

- The compiler shows the line number where it found the error.
- The error messages can be tough to understand!

Strings

- **string**: A sequence of characters.
 - Starts and ends with a " quote " character.
 - The quotes do not appear in the output.
 - Examples:
`"hello"`
`"This is a string. It's very long!"`
- Restrictions:
 - May not span multiple lines.
`"This is not
a legal String."`
 - May not contain a " character.
`"This is not a "legal" String either."`

Escape sequences

- **escape sequence:** A special sequence of characters used to represent certain special characters in a string.

<code>\t</code>	tab character
<code>\n</code>	new line character
<code>\"</code>	quotation mark character
<code>\\</code>	backslash character

- **Example:**

```
System.out.println("\\hello\\nhow\\tare  \"you\"?\\\\\\");
```

- **Output:**

```
\\hello
how      are  "you"?\\
```

Questions

- What is the output of the following `println` statements?

```
System.out.println("\ta\tb\tc");  
System.out.println("\\\\");  
System.out.println("'");  
System.out.println("\"\"");  
System.out.println("C:\nin\the downward spiral");
```

- Write a `println` statement to produce this output:

```
/ \ // \\ /// \\\
```

Answers

- Output of each `println` statement:

```
          a          b          c
\\
'
""
C:
in          he downward spiral
```

- `println` statement to produce the line of output:

```
System.out.println("/  \\  //  \\\\  ///  \\\\\\\");
```

Comments

- **comment:** A note written in source code by the programmer to describe or clarify the code.
 - Comments are not executed when your program runs.
- Syntax:
 - // comment text, on one line**
 - or,
 - /* comment text; may span multiple lines */**
- Examples:
 - // This is a one-line comment.**
 - /* This is a very long
multi-line comment. */**

Using comments

- Where to place comments:
 - at the top of each file (a "comment header")
 - at the start of every method (seen later)
 - to explain complex pieces of code
- Comments are useful for:
 - Understanding larger, more complex programs.
 - Multiple programmers working together, who must understand each other's code.

Static methods

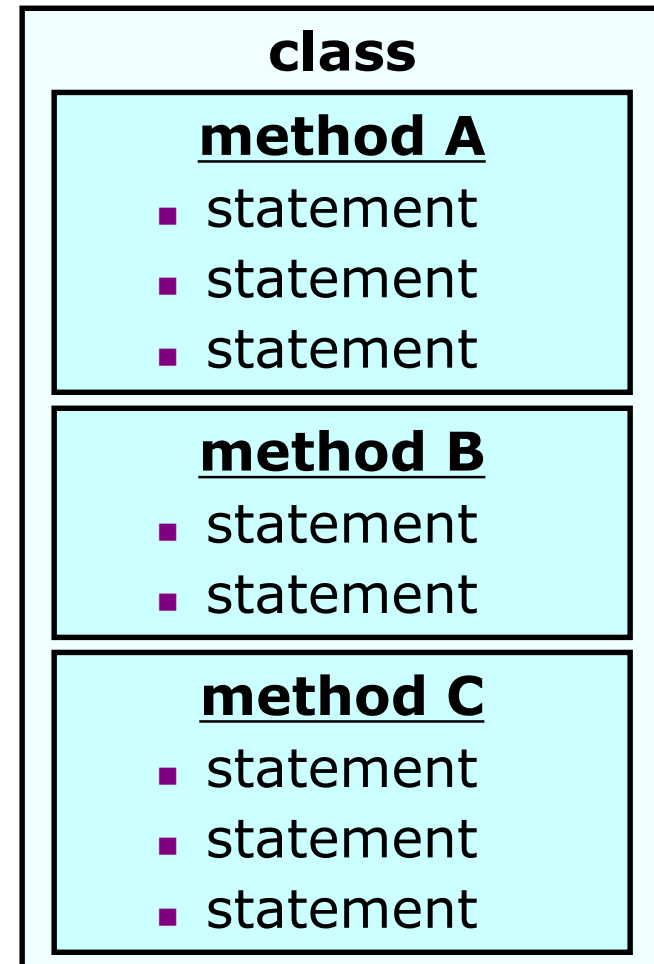
Static methods

- **static method:** A named group of statements.

- denotes the *structure* of a program
- eliminates *redundancy* by code reuse

- **procedural decomposition:**
dividing a problem into methods

- Writing a static method is like
adding a new command to Java.



Using static methods

1. Design the algorithm.
 - Look at the structure, and which commands are repeated.
 - Decide what are the important overall tasks.
2. **Declare** (write down) the methods.
 - Arrange statements into groups and give each group a name.
3. **Call** (run) the methods.
 - The program's `main` method executes the other methods to perform the overall task.

Declaring a method

Gives your method a name so it can be executed

- Syntax:

```
public static void name() {  
    statement;  
    statement;  
    ...  
    statement;  
}
```

- Example:

```
public static void printWarning() {  
    System.out.println("This product causes cancer");  
    System.out.println("in lab rats and humans.");  
}
```

Calling a method

Executes the method's code

- Syntax:

name () ;

- You can call the same method many times if you like.

- Example:

```
printWarning ( ) ;
```

- Output:

```
This product causes cancer  
in lab rats and humans.
```

Methods calling methods



```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Done with main.");  
    }  
  
    public static void message1() {  
        System.out.println("This is message1.");  
    }  
  
    public static void message2() {  
        System.out.println("This is message2.");  
        message1();  
        System.out.println("Done with message2.");  
    }  
}
```

- **Output:**


```
This is message1.  
This is message2.  
This is message1.  
Done with message2.  
Done with main.
```

Control flow

- When a method is called, the program's execution...
 - "jumps" into that method, executing its statements, then
 - "jumps" back to the point where the method was called.

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1 () ;   
        message2 () ;   
        System.out.println("Done with message1.");  
    }  
    ...  
}
```

```
public static void message1() {  
    System.out.println("This is message1.");  
}
```

```
public static void message2() {  
    System.out.println("This is message2.");  
    message1 () ;   
    System.out.println("Done with message2.");  
}
```

```
public static void message1() {  
    System.out.println("This is message1.");  
}
```

When to use methods

- Place statements into a static method if:
 - The statements are related structurally, and/or
 - The statements are repeated.
- You should not create static methods for:
 - An individual `println` statement.
 - Only blank lines. (Put blank `println`s in `main`.)
 - Unrelated or weakly related statements.
(Consider splitting them into two smaller methods.)