Java Programming Language

History of Java

Initial days:

- Started as an internal project at Sun Microsystems in 1992
- Headed by James Gosling, Patrik Naughton, and Mike Sheridan
- Was meant to be used for small embedded systems
- Was called initially as GreenTalk by Gosling with file extension of .gt
- Later was called as Oak and was developed as part of the Green Project
- In 1995, it was renamed as Java, since Oak was already registered by Oak Technologies
- Java 1.0a released for download
- JDK 1.0 was released on 23rd January 1996

Version history:

- 1995 - JDK Alpha and beta
- 23rd Jan 1996 - JDK 1.0
- 19th Feb 1997 - JDK 1.1
- 8th Dec 1998 - J2SE 1.2
- 8th May 2000 - J2SE 1.3
- 6th Feb 2020 - J2SE 1.4
- 30th Sep 2004 - J2SE 1.5 (Java 5)
- 11th Dec 2006 - Java SE 6
- 28th Jul 2011 - Java SE 7
- 18th Mar 2014 - Java SE 8

- Java community process
- http://jcp.org/
- The Java Community Process
- Java Specification Requests (JSRs)
- The Java Language Specification (JLS)

Getting started…

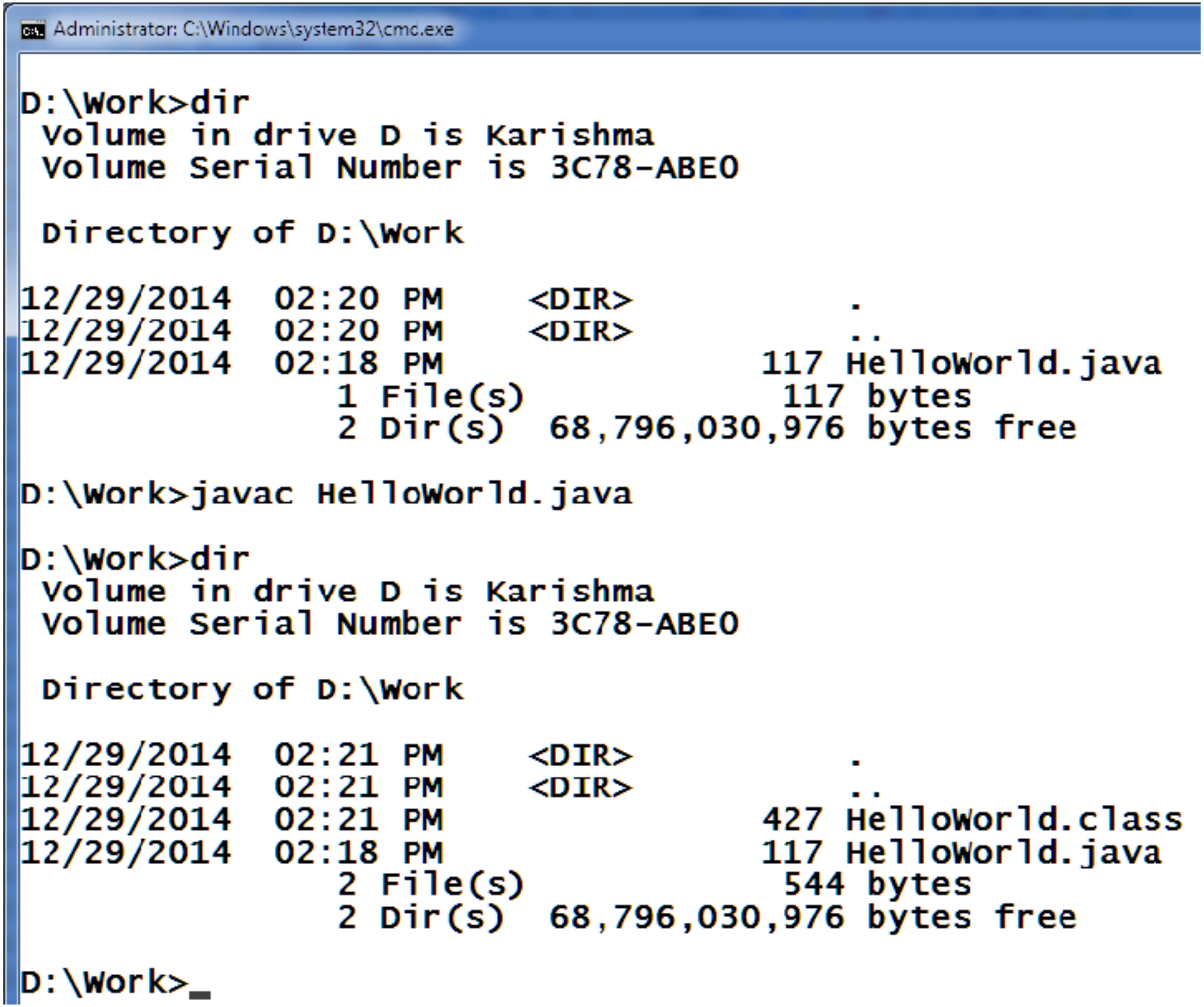
Download and install JDK for your OS

- http://oracle.com/technetwork/java/javase/downloads/
- Add **%JAVA_HOME%\bin** to the OS Path
- Test using command prompt:
  - java -version
  - javac -version

```
// HelloWorld.java

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

Compiling the source code:

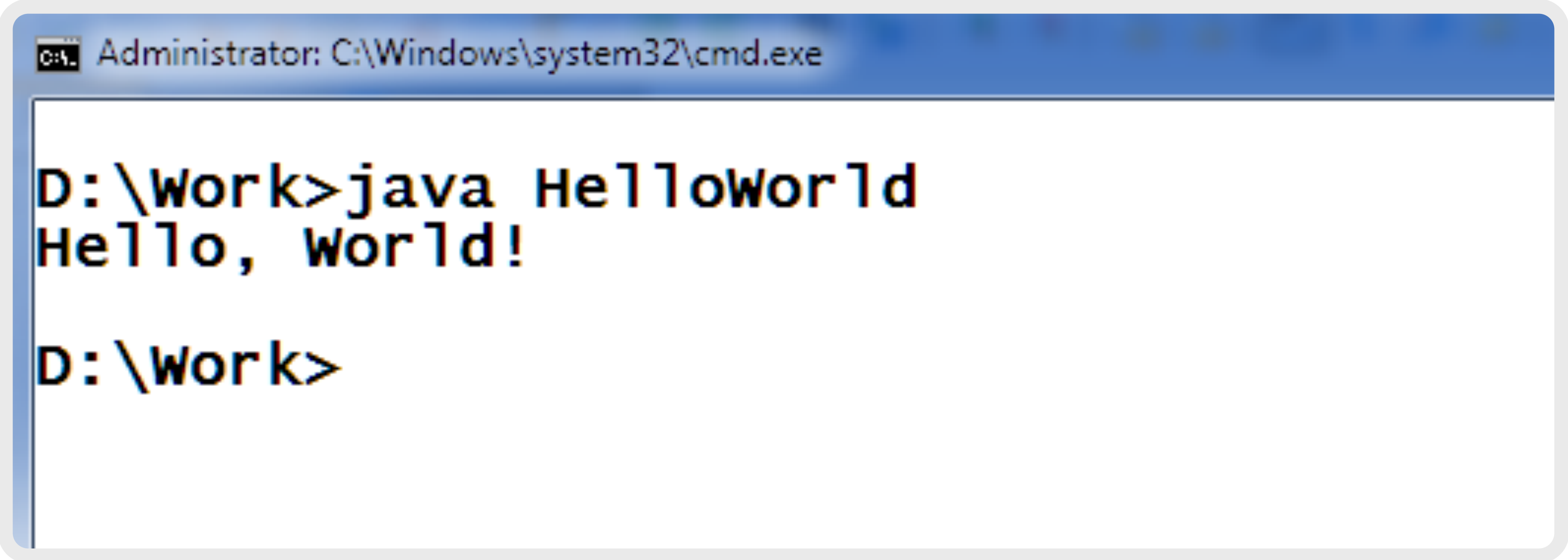
- Use the javac.exe (or simply “javac”)
- Generates one .class file for each of the class definition in the source code
- Keeps in the same folder as the source
- Can specify a different location



```
D:\Work>dir
 Volume in drive D is Karishma
 Volume Serial Number is 3C78-ABE0

 Directory of D:\Work

12/29/2014  02:20 PM    <DIR>          .
12/29/2014  02:20 PM    <DIR>          ..
12/29/2014  02:18 PM               117 HelloWorld.java
               1 File(s)            117 bytes
               2 Dir(s)  68,796,030,976 bytes free

D:\Work>javac HelloWorld.java

D:\Work>dir
 Volume in drive D is Karishma
 Volume Serial Number is 3C78-ABE0

 Directory of D:\Work

12/29/2014  02:21 PM    <DIR>          .
12/29/2014  02:21 PM    <DIR>          ..
12/29/2014  02:21 PM               427 HelloWorld.class
12/29/2014  02:18 PM               117 HelloWorld.java
               2 File(s)            544 bytes
               2 Dir(s)  68,796,030,976 bytes free

D:\Work>_
```

Executing the generated class

- Use the java.exe (or simply "java")
- Takes the name of the class (no extension) as argument
- The class must contain a "main" function, which is public, static, and void



```
D:\Work>java HelloWorld
Hello, World!

D:\Work>
```

Variable

- A storage location paired with a symbolic name (identifier).
- A.K.A. Scalar
- Contains some known or unknown quantity or information:
    - Referred to as a value
    - Can be changed during the program execution

Variable

- The variable name is the usual way to reference the stored value.
- In Java, no access to the actual address, unlike C or C++.
- In Java, there are two types of variables:
  - Primitives
  - References

Primitives

Variables of built-in Java's core data types:

- Integers
  - byte, short, int, long
- Decimals
  - float, double
- Character
  - char
- Boolean
  - boolean

Primitives

- Static memory allocation
- Size of variable depends on the data type

Example:

- The size of a “char” is 2 bytes and size of “double” is 8 bytes.

References

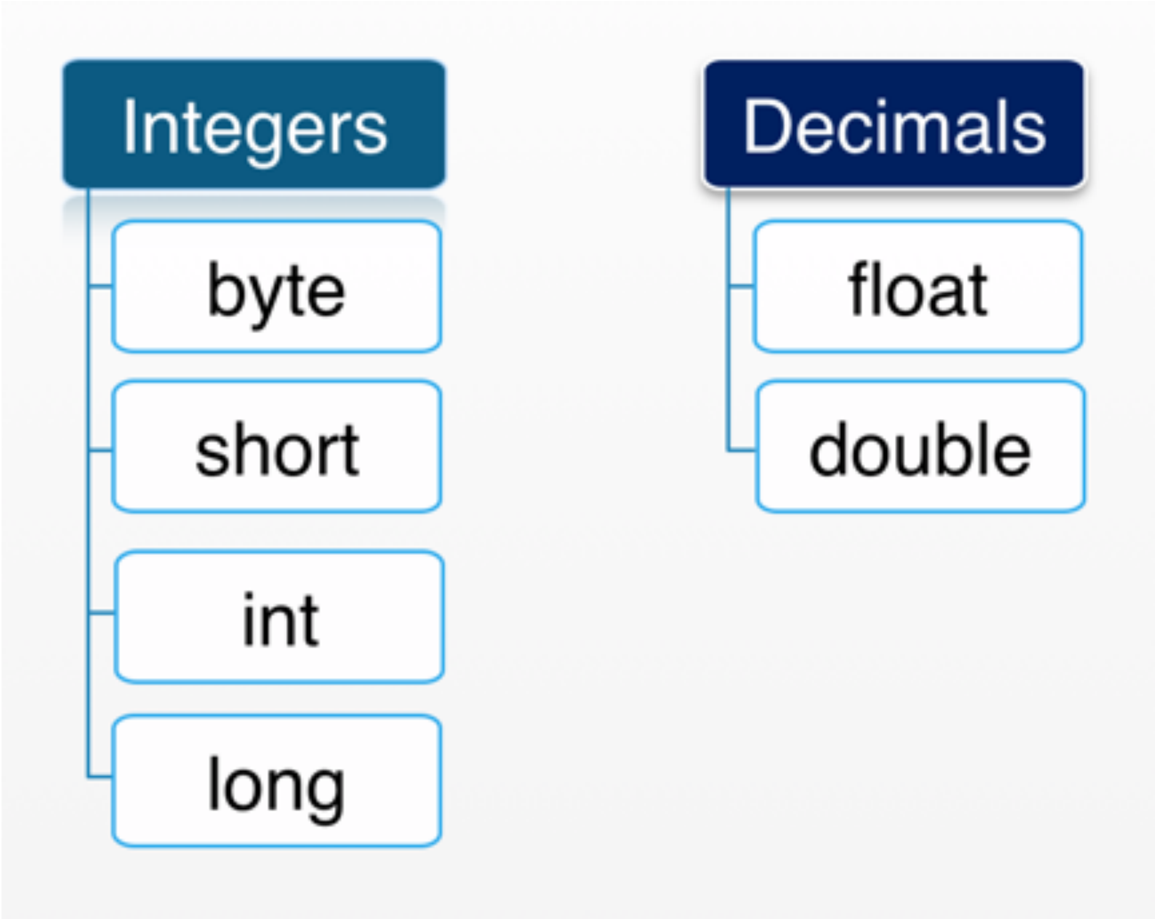
- Declared using a Class, Interface, Enum, or Arrays
- The size of a reference is fixed and does not depend on the type used for creating it
- Initialized to a reference number representing an object
- Can be assigned with "null", when not referencing to an object
- Not same as pointers in C/C++
- Can be used as method arguments and return types
- Can be assigned to another reference of similar type
- Can not be used with operators such as arithmetic or relational operators

Creating and initializing references

```
Person p1, p2;
p1 = new Person();
p1.age = 44;
p2 = p1;
p1 = null;
p2 = null;
```

Java is statically typed

- All variables must be declared before using
- Data type variable [= initialValue];



Integers store the binary equivalent of the number.

- 307
- 0000 0001 0011 0011

Negative numbers are stored in 2's complement format.

- -310
- 0000 0001 0011 0110 (unsigned)
- 1111 1110 1100 1001 (1's complement)
- 1111 1110 1100 1010 (2's complement)

Integers

- byte (1 byte, -128 to 127)
- short (2 bytes, -32768 to 32767)
- int (4 bytes, -2147483648 to 2147483647)
- long (8 bytes, -9223372036854775808 to 9223372036854775807)

Preference

- "int" is preferred.
- Java compilers and runtimes are tuned to work with int.
- In the heap, all integers have a default value of 0.

Decimals

- float (4 bytes, 1.4E-45 to 3.4028235E38)
- double (8 bytes, 4.9E-324 to 1.7976931348623157E308)

Preference

- "double" is preferred.
- Java compilers and runtimes are tuned to work with double.
- In the heap, all decimals have a default value of 0.0.

Boolean

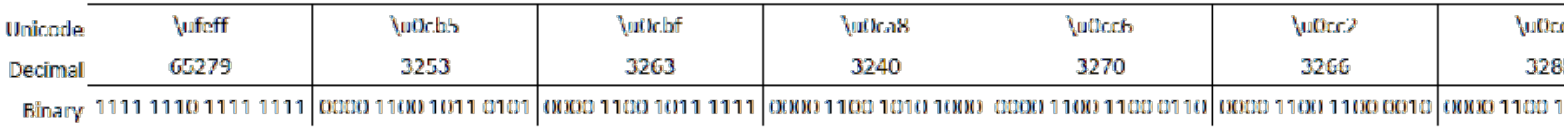
- Possible values: true and false
- Use this data type for simple flags that track true/false conditions
- This data type represents one bit of information
- All relational operations result in boolean
- When printed, literals "true" or "false" are used
- In the heap, a boolean variable has a default value of "false"

Characters

- char
- 2 bytes
- Stores Unicode corresponding to a character
- Standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems

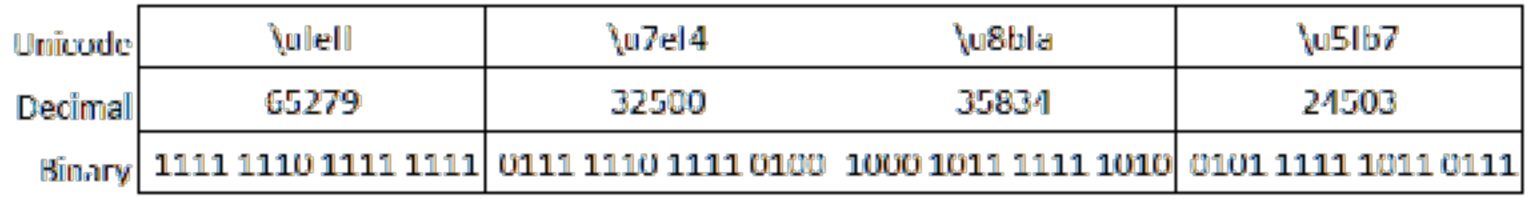
Unicode

ವಿನೋದ್



Unicode	\ufeff	\u0cb5	\u0cbf	\u0ca8	\u0cc6	\u0cc2	\u0c
Decimal	65279	3253	3263	3240	3270	3266	328
Binary	1111 1110 1111 1111	0000 1100 1011 0101	0000 1100 1011 1111	0000 1100 1010 1000	0000 1100 1100 0110	0000 1100 1100 0010	0000 1100 1

维诺德

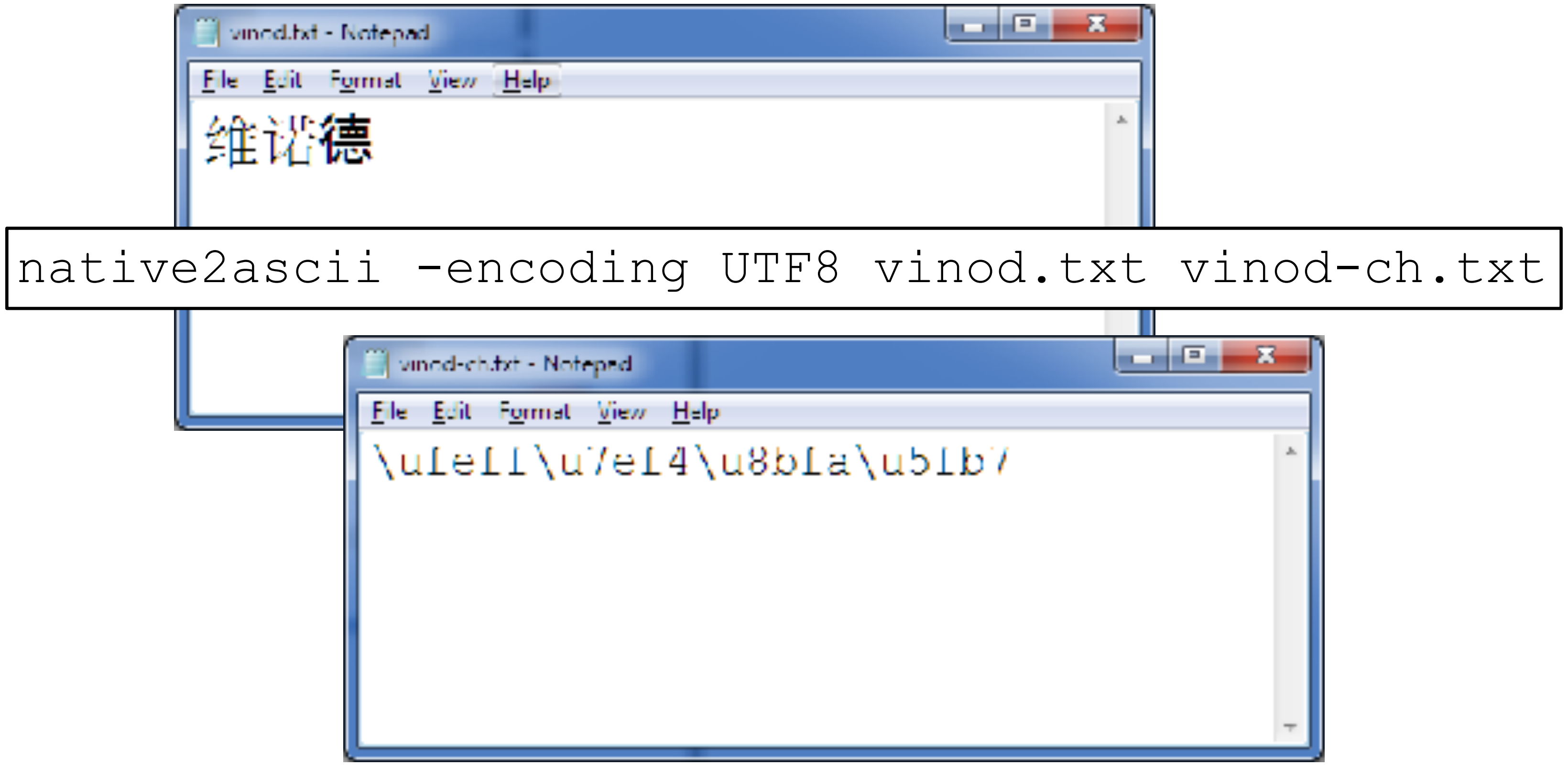


Unicode	\ufeff	\u7ef4	\u8bfa	\u5fb7
Decimal	65279	32500	35834	24503
Binary	1111 1110 1111 1111	0111 1110 1111 0100	1000 1011 1111 1010	0101 1111 1011 0111

© https://vinod.co

Unicode conversion

- JDK comes with a tool called native2ascii
- Converts a file containing native language text to Unicode



维诺德

```
native2ascii -encoding UTF8 vinod.txt vinod-ch.txt
```

```
\ufeff\u7ef4\u8bfa\u5fb7
```

Operators

- Operators perform on single or multiple operands to return a result.
- Operators are special symbols
- Perform specific operations on one, two, or three operands
- Expressions return a result

Types of operators:

- Unary
- Binary
- Ternary

Unary operators:

- Work on a single operand:
- Postfix expr++ expr--
- Prefix ++expr --expr
- Sign indicator +expr -expr
- Bitwise Not ~
- Logical Not !

Binary operators:

- Work on two operand on either side of the operator:
- Arithmetic: + - * / %
- Relational: > >= < <= == !=
- Logical: && ||
- Bitwise: << >> >>> & | ^

Ternary operator

- Works with 3 operands
- bool_expr ? true_expr : false_expr

- Examples:

```
int big = a>b ? a: b;
int big = (a>b && a>c) ? a : (b>c ? b: c);
int maxDays = isLeap(year) ? 366 : 365;
```

Assignment operators:

- =   +=   -=   *=   /=   %=
- <<=   >>=   >>>=
- &= |=   ^=

Programming constructs

if-else construct

- "if-else" condition is represented as a Boolean expression.
- "else" construct is used to execute a code when the condition in the “if” construct fails.
- Motivation:
  - Execute code based on certain conditions
  - A condition is represented as a boolean expression

Syntax:

```
if(condition)
   statement;

if(condition){
   statement1;
   statemenT2;
   ...
   statementN;
}
```

Example:

- Check if a number is even.

```
int num = ...

if((num%2) == 0){
    System.out.println("Even");
}
```

When it fails...

- Use the "else" construct to execute a code, when the condition in the "if" construct fails
- Use of "else" is optional
- Only one "else" per "if"
- "else" must immediately follow an "if"

Example:

Check if a number is even or odd.

```
int num = ...
if( (num%2) == 0 ){
    System.out.println("Even");
}
else{
    System.out.println("Odd");
}
```

Best practices

- Always use curly braces
  - Improves readability and maintenance

Multiple conditional executions

- More than one alternate:

```
if(condition1)
    statement1;
else if(condition2)
    statement2;
...
   ...
else if(conditionN)
    statementN;
else
    default-statement
```

Multiple conditional executions

- Assuming 28 days in February:

```
int days;
if(month==2){
    days = 28;
}
else if(month==4 || month==6
    || month==9 || month==11){

    days = 30;
}
else {
    days = 31;
}
```

Best practice

- Avoid unnecessary "if" or "else":

```
int days = 31;

if(month==2){
    days = 28;
}
else if(month==4 || month==6
    || month==9 || month==11){

    days = 30;
}
```

Nesting "if" constructs

```
int days = 31;

if(month==2){
    days = 28;

    if(year%4==0 && year%100!=0 || year%400==0){
        days = 29;
    }
}
else if(month==4 || month==6  || month==9 || month==11){
    days = 30;
}
```

Avoid when possible

```
boolean isLeap(int year){
    if(year%4==0 && year%100!=0 || year%400==0){
        return true;
    }
    else {
        return false;
    }
}
```

```
boolean isLeap(int year){
    return (year%4==0 && year%100!=0 || year%400==0);
}
```

The "switch-case" construct:

- A type of selection control mechanism is used to allow the value of a variable or expression to change the control flow of program execution via a multiway branch
- Improves clarity by reducing repetitive coding
- Faster execution through easier compiler optimization

Syntax:

```
switch(expression){
    case value :
       //Statements
       break;
    case value :
       //Statements
       break;
    default :
       //Statements
}
```

Example:

```
int days;

switch(month){
   case 2:
      days = 28;
      break;
   case 4:
   case 6:
   case 9:
   case 11:
      days = 30;
      break;
   default:
      days = 31;
}
```

switch-case can not use boolean expressions with "case".

```
switch(someNumber){
   case > 10000:
      System.out.print("Too big");
   case > 5000:
      System.out.print("Moderate");
   default:
      System.out.print("Too small");
}
```

switch can't take float, double, boolean

```
boolean tf = true;

switch(tf){
   case true:
      System.out.println("True");
     break;
   case false:
      System.out.println("False");
}
```

From Java 7 onwards:

- 'swtich' can take a String variable
- Value comparison (not reference)
- String comparison is case sensitive

Restriction:

- case expr must be a constant

```
switch("Vinod"){
    case name1:
        System.out.println("name1 is Vinod");
        break;
    case name2:
        System.out.println("name1 is Vinod");
        break;
    case name3:
        System.out.println("name1 is Vinod");
        break;
}
```

When to use?

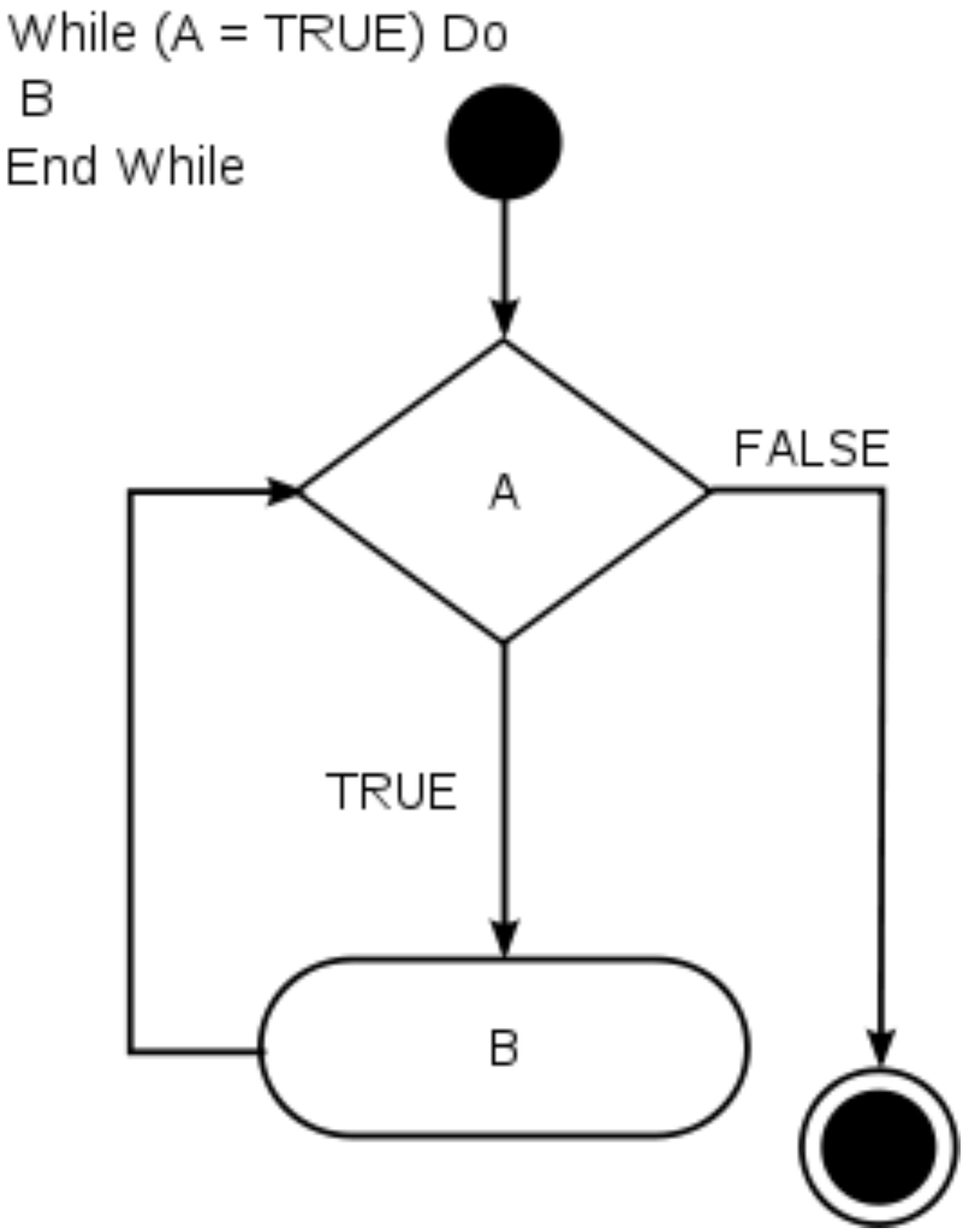
- Prefer using switch-case:
  - to improve readability
  - to execute a common code, if a variable matches one of many values

The "while" loop

- Allows code to be executed repeatedly based on a given Boolean condition
- Can be thought of as a repeating “if” statement

The "while" loop

- It consists of a block of code and a condition.
- Condition is evaluated, and if the condition is true, the code within the block is executed.
- This repeats until the condition becomes false.
- a.k.a. pre-test loop.



Example:

```
int sum = 0;
int i = 1;

while(i<=5){
   sum = sum + i;
   i++;
}

System.out.println(sum);
```

Boolean always!!

- Unlike the C or C++ language, the while construct always require a boolean expression (or value).
- The following would be an error:

```
int i = 10;

while(i){
   System.out.println(i--);
}
```

Infinite loop:

- Supplying a "true" to the while construct makes it an endless (infinite) loop.
- Exit the loop using "break".

```
int i = 10;

while(true){
   System.out.println(i--);
   if(i==0) break;
}
```

The "for" loop

- Most commonly used loop
- Number of iterations is known in advance

Syntax:

```
for(expr1; expr2; expr3) {
   // loop statements
}
```

- Initializer
  - int i=0
- Criteria
  - i<10
- Loop controller
  - i++

Example:

- Factorial of a number

```
int f = 1;

for(int i=2; i<=5; i++){
   f = f * i;
}

System.out.println(f);
```

Things to avoid

```
for(int i=0; i < list.size(); i++){
   // do something
}
```

- list.size() is a function call
- Gets executed for each iteration
- Results in low performance

Things to avoid

```
for(int i=0, j=list.size(); i<j; i++){
   // do something
}
```

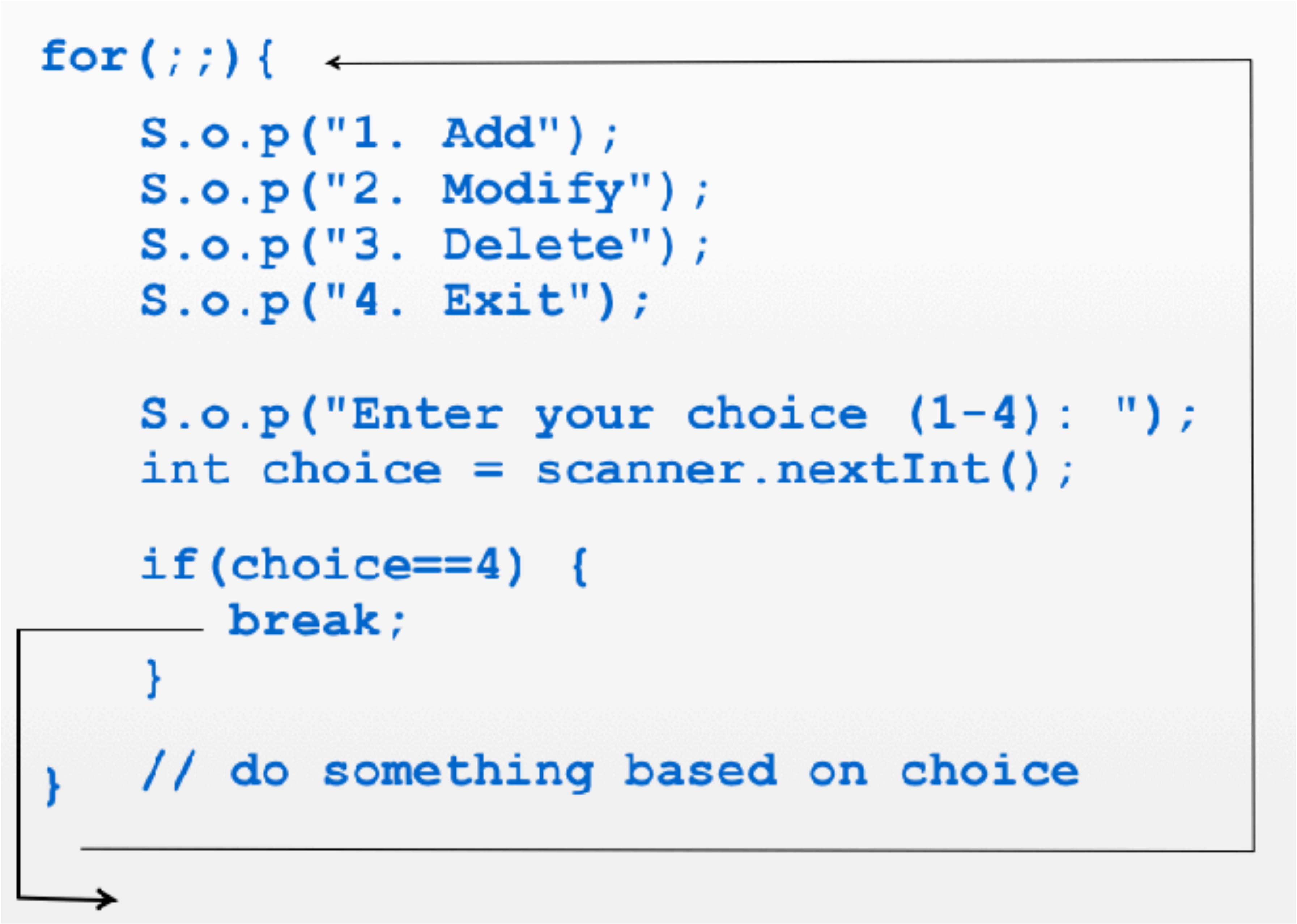
- Declare a variable to contain the list.size() in the initializer section
- Gets executed only once
- Increases the performance

Infinite loop

- The loop that never ends on its own
- Use break, return, throw or System.exit(n) to stop the loop

```
for(;;){
   // do something
}
```

Example of an infinite loop:



```
for(;;){
    S.o.p("1. Add");
    S.o.p("2. Modify");
    S.o.p("3. Delete");
    S.o.p("4. Exit");

    S.o.p("Enter your choice (1-4): ");
    int choice = scanner.nextInt();

    if(choice==4) {
      break;
    }

    // do something based on choice
}
```

The "do-while" loop

- Post checked while loop
- Loop body is executed at least once
- Subsequent iterations depend on the loop criteria
- Useful when you process a menu selection or input validation

```
do {
   // statements;
} while(expr);
```

Example:

```
do {

   S.o.p("Enter month (1-12): ");
   month = scanner.nextInt();

   boolean isValid = month >=1 && month <=12;

   if(!isValid){
      S.o.p("Invalid month!");
   }

}while(!isValid);
```

The do-while loop is:

- The least preferred loop
- Generally used for simple menus and user input validations
- Used when the loop criteria is not known in the beginning of the loop
- Alternate to this, we may also use an infinite while/for loop, with a force-exit at the end of the loop.

Alternates:

```
while(true){
   // loop statements

   if(some_criteria){
      break;
   }
}
```

```
for(;;){
   // loop statements

   if(some_criteria){
      break;
   }
}
```