Object-Oriented Programming

Java Programming Essentials

United International College

Review

Can you write HelloWorld without looking at the code now?

Java Programming Essentials

- **Keywords**
- **Variable Declarations and Assignments**
- **Standard input and output (I/O)**
- **Data Types**
- **Type Casting**
- **Arithmetic Expressions and Operators**
- **Java Packages**
- **Flow Control**

Punctuations

- ' Single Quote
- " Double Quote
- () Brackets/ Parentheses
- [] Square Brackets
- {} Curly Brackets/ Braces
- ; Semi-colon
- . Dot/ Full-stop
- , Comma
- / slash
- \ back slash

Identifiers

The name of a variable or other item (class, method, object, etc.) defined in a program.

- A Java identifier must NOT start with a digit, and all the characters must be letters, digits, "`$`" or "`_`"
  - `W_12`, `HelloWorld`, `_983`, `$bS5_c7` Correct
  - `4W2`, `class`, `Data#`, `98.3`, `Hell world` Not Ok

- Java is a **case-sensitive** language: `Rate`, `rate`, and `RATE` are the names of three different variables

Choice of Identifiers

- Easy to understand and remember.
  - e.g. `numberOfEnquiries`, `Trees`, `timeToLive`, `name`, `address`, `isOK`
- May use multiple words (without space!).
  - e.g. `myBirthday`, `numberOfStudents`
- Don't be lazy!
  - e.g. `i`, `j`, `k`, `a`, `b`, `c`, `d`, `e`, ...

Keywords

Keywords and **Reserved** words: These are identifiers having a predefined meaning in Java

- Do not use them to name anything else

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

Java Library Identifiers

Java Library Identifiers: Defined in libraries required by the Java language standard

- Although they can be redefined, this could be confusing and dangerous if doing so since it would change their standard meaning

```
System    String    println
```

Naming Conventions

Camel Case

- **Variable and Method** names should begin with a lowercase letter. Indicate "word" boundaries with an uppercase letter and restrict the remaining characters to digits and lowercase letters

  `topSpeed` `bankRate1` `timeOfArrival`

- **Class** names should begin with an uppercase letter and, otherwise, adhere to the rules above

  `FirstProgram` `MyClass` `String`

Variable Declarations

```
int numberOfBeans;
double myBalance, totalWeight;
```

- Every variable in a Java program must be ***declared*** before it is used
  - A variable declaration tells the compiler what kind of data (type) will be stored in the variable
  - The type of the variable is followed by one or more variable names separated by commas, and terminated with a semicolon
  - Variables are typically declared just before they are used or at the start of a block (indicated by an opening brace `{`)
  - Basic types in Java are called *primitive types*

Variable Declarations

```
String myName = "Chunyan Ji";
double myHeight = 1.62, rainfall = 3.4;
```

- "Type" refers to both *primitive type* or *class*.
- In general, four basic forms of declarations:

```
1.<type name> identifier;
2.<type name> identifier1, identifier2, … ;
3.<type name> identifier = <initial value>;
4.<type name> identifier1 = <initial value1>,
        identifier2 = <initial value2>, …;
```

- Tip: always initialize the variable, set an initial value!

Three kinds of variables

- There are three kinds of variables in Java
  - Local variables are declared (visible) in methods, constructors or blocks. No default value for local variables, therefore should be assigned an initial value before the first use.
  - Instance variables are declared in a class, but outside a method, constructor or any block. Instance variables have default values, e.g., numbers – 0, Boolean – false, etc.
  - Class/Static variables are declared with the static keyword in a class, but outside a method, constructor or blocks. There would only be one copy of each class variable per class, regardless of how many objects are created from it. Default values are same with instance variables.

Local Variables

```
public class Test {
        public void pupAge()
        {
                int age = 0;
                age = age + 7;
                System.out.println("Puppy age is : " + age);
        }
        public static void main(String args[]) {
                Test test = new Test();
                test.pupAge();
        }
}
```

- If we use variable age without initializing it, e.g., "int age;" it would give an error at the time of compilation.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The local variable age may not have been initialized

        at Test.pupAge(Test.java:5)
        at Test.main(Test.java:11)
```

Instance Variables

```
public class Test {
        int age;
        public void pupAge()
        {
                age = age + 7;
                System.out.println("Puppy age is : " + age);
        }
        public static void main(String args[])
        {
                Test test = new Test();
                test.pupAge();
        }
}
```

- The output is "Puppy age is: 7", the variable age is an instance variable with default value of 0.

Class/Static Variables

```
public class Test {
        static int age;
        public void pupAge()
        {
                age = age + 7;
                System.out.println("Puppy age is : " + age);
        }
        public static void main(String args[])
        {
                Test test1 = new Test();
                test1.pupAge();
                Test test2 = new Test();
                test2.pupAge();
        }
}
```

- What is the output result?
- What is the output result if we change to "int age;"?

Standard input and output (I/O)

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen.

Standard output

Standard Output – This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**

```
System.out.println();  //Print a line to the standard output (screen)
System.out.print();    //Print something to the standard output (screen)
```

Standard output: example 1

```
public class HelloWorld {

  /* The HelloWorld Program
     -------------------
     Illustrates a simple program displaying
     a message.
  */

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

Output:

```
HelloWorld!
```

Standard output: example 2

```
public class HelloWorld {

  /* The HelloWorld Program
     -------------------
     Illustrates a simple program displaying
     a message.
  */

  public static void main (String[] args) {
    System.out.print("HelloWorld!");
  }
}
```

Output:

```
HelloWorld!
```

Standard output: example 3

```
public class HelloWorld {

  /* The HelloWorld Program
     -------------------
     Illustrates a simple program displaying
     a message.
  */

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

Output:

```
HelloWorld!
HelloWorld!
```

Standard output: example 4

```
public class HelloWorld {

  /* The HelloWorld Program
     -------------------
     Illustrates a simple program displaying
     a message.
  */

  public static void main (String[] args) {
    System.out.print("HelloWorld!");
    System.out.print("HelloWorld!");
  }
}
```

Output:

```
HelloWorld!HelloWorld!
```

Standard input

- **Scanner**, as the name implied, is a simple text **scanner** which can parse the input text into primitive types and strings using regular expressions.
- It first breaks the text input into tokens using a delimiter pattern, which is by default the white spaces (blank, tab and newline).
- The tokens may then be converted into primitive values of different types using the various **nextXxx**() methods (**nextInt**(), **nextByte**(), **nextShort**(), **nextLong**(), **nextFloat**(), **nextDouble**(), **nextBoolean**(), **next**() for String, and **nextLine**() for an input line).
- You can also use the **hasNextXxx**() methods to check for the availability of a desired input.

```
import java.util.Scanner;              //we need to use the Scanner class
Scanner input1 = new Scanner(System.in);    //declare a new Scanner object
int i = input1.nextInt();                   //input an integer from screen
Double d = input1.nextDouble();        //input an double variable from screen
String s=input1.next();                     //input a string variable from screen
```

Standard input: example 1

```
import java.util.Scanner;

public class Hello {

  public static void main (String[] args) {
    Scanner input1 = new Scanner(System.in);
    System.out.println("Please enter your name:");
    String s=input1.next();
    System.out.println("Hello "+s+"!");
  }
}
```

Output:

```
Please enter your name:
xyz
Hello xyz!
```

Standard input: example 2

```
import java.util.Scanner;

public class Sum {

  public static void main (String[] args) {
    Scanner input1 = new Scanner(System.in);
    System.out.println("Please input integer a:");
    int a = input1.nextInt();
    System.out.println("Please input integer b:");
    int b = input1.nextInt();
    int c = a+b;
    System.out.println("The sum of a and b is "+c);
    }
}
```

Output:

```
Please input integer a:
10
Please input integer b:
20
The sum of a and b is 30
```

What is a Type?

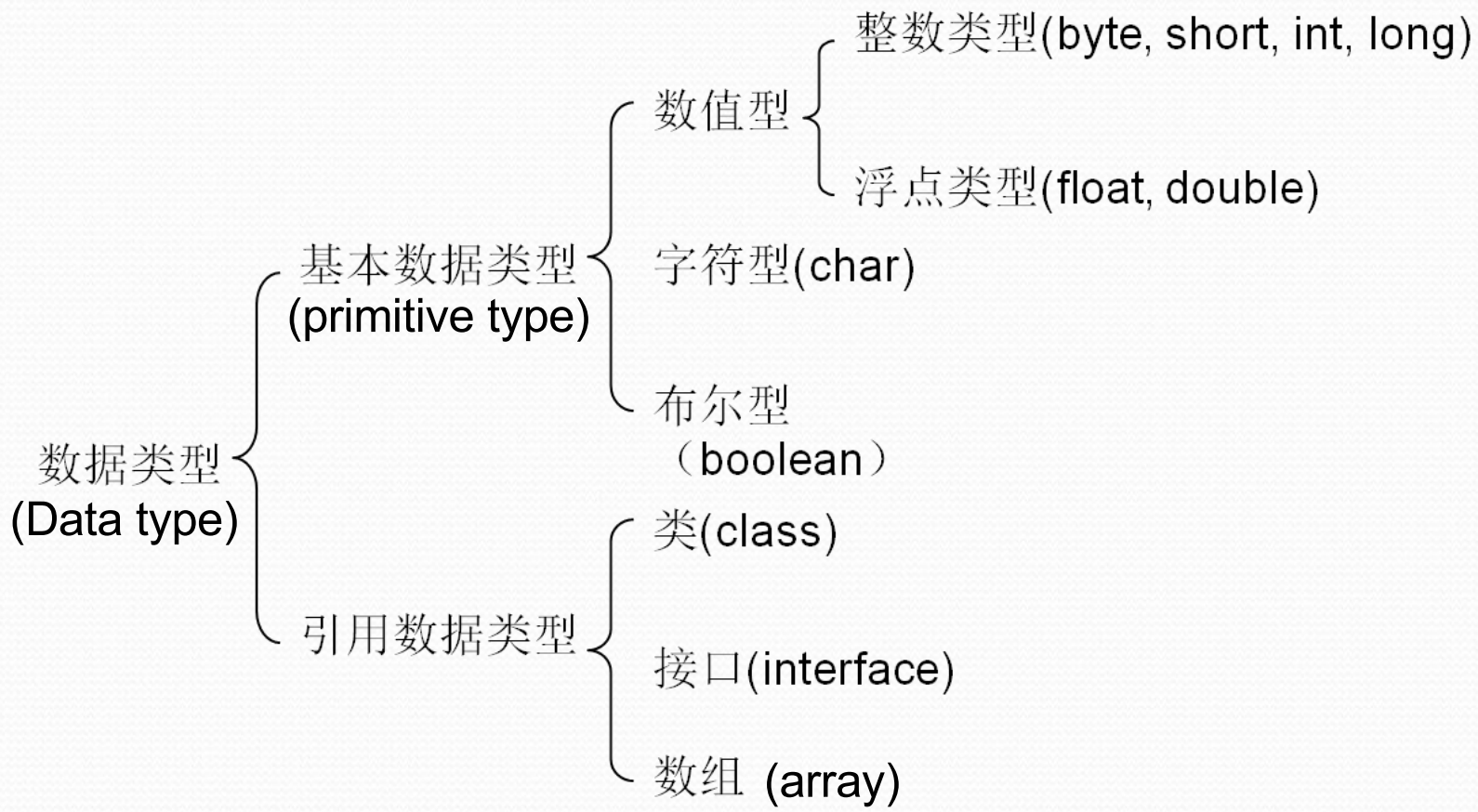
- A ***type*** restricts the **kind and range of values** a data item or an expression could take.
  - e.g. Sex → ['M', 'F']
    Phone → 13211113231

- Type must be matched
  - e.g.
```
String name = "Lily Lin";
String myID = "123";
```

123 is an **integer**

"123" is a **String**

Data Types


数据类型
(Data type)
基本数据类型
(primitive type)
数值型
整数类型(byte, short, int, long)
浮点类型(float, double)
字符型(char)
布尔型
（boolean）
引用数据类型
类(class)
接口(interface)
数组 (array)

Primitive Types

Java defined Four categories and Eight different primitive types:

- Boolean: **`boolean`**
- Character: **`char`**
- Integer: **`byte`**, **`short`**, **`int`**, **`long`**
- Floating point: **`float`**, **`double`**

Primitive Types

Data Type	Default Value (for fields)	Size (in bits)	Minimum Range	Maximum Range
byte	0	Occupy 8 bits in memory	-128	+127
short	0	Occupy 16 bits in memory	-32768	+32767
int	0	Occupy 32 bits in memory	-2147483648	+2147483647
long	0L	Occupy 64 bits in memory	-9223372036854775808	+9223372036854775807
float	0.0f	Occupy 32-bit IEEE 754 floating point	1.40129846432481707e-45	3.40282346638528860e+38
double	0.0d	Occupy 64-bit IEEE 754 floating point	4.94065645841246544e-324d	1.79769313486231570e+308d
char	'\u0000'	Occupy 16-bit, unsigned Unicode character		0 to 65,535
boolean	false	Occupy 1- bit in memory	NA	NA

Boolean

- The **`boolean`** type is used for evaluating logical conditions. Usually used in flow of control
- Has two values: **`false`** and **`true`**.

```
boolean flag;
flag = true;
if(flag) {
        // do something
}
```

Character

- A **`char`** variable stores a single character
- Character literals are delimited by single quotes:
  - `'a'  'X'  '7'  '$'  ','  '\n'`
- Example declarations:
  - `char topGrade = 'A';`
  - `char terminator = ';';`

Do you see the difference between **`char`** and **`String`**?

`'A' does not equal to "A"`

Character

Escape Sequence	Name	Unicode Value
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000a
\r	Carriage return	\u000d
\"	Double quote	\u0022
\'	Single quote	\u0027
\\	Backslash	\u005c

Integer

- The **`integer`** types are for numbers without factional parts. Negative values are allowed.
- Long integer numbers have a suffix '**`L`**' or '**`l`**'
  - **`int i = 600;`** // correct
  - **`long l = 88888888881;`** // wrong without suffix '`l`'

byte	integer	1 byte	−128 to 127
short	integer	2 bytes	−32768 to 32767
int	integer	4 bytes	−2147483648 to 2147483647
long	integer	8 bytes	−9223372036854775808 to 9223372036854775807

Integer

- Decimal number:
  - E.g.: `12`, `-23`, `0`
- Octal number:
  - Begin with "`0`". E.g.: `012`
- Hexadecimal number:
  - Begin with "`0X`" or "`0x`". E.g.: `0x12`

Floating numbers

Type	Storage Requirement	Range
`float`	4 bytes	approximately ±3.40282347E+38F (6–7 significant decimal digits)
`double`	8 bytes	approximately ±1.79769313486231570E+308 (15 significant decimal digits)

Floating numbers

- Floating-point Number literals are considered to be of type **`double`** by *default*.

```
double d1 = 3.14159;                    // ok
double d2 = 3e8;                        // ok
double d3 = -0.27e-5;                   // ok
float  f1 = 3.14159;                    // not ok
```

- Adding a suffix F/ f to the number changes this default:

```
float  f2 = 3.14159F;                   // ok
float  f3 = -0.27e-5f;                  // ok
```

Default Type of numbers (int and double)

- ***Integer literals*** (integer numbers appearing in program code) are generally treated as **`int`** type.
  - e.g. `i = 7 * j / (-9 + k);`
  - e.g. `Math.cos(30 * Math.PI / 180);`
- ***Real number/ floating point number literals*** (numbers with decimal point or in exponential notation appearing in program code) are generally treated as **`double`** type.
  - e.g. `p = 7.0 * q + .958;`
  - e.g. `Math.toRadians(3e-1 * 100.);`

3.0×10^{-1}

Example: Initial variables

After you declare a variable, you must explicitly initialize it by means of an assignment statement—you can never use the values of uninitialized variables.

```
public class PrimaryType {
        public static void main (String args []) {
                boolean b = true;       //boolean type
                int x, y=8;             // int type
                float f = 4.5f;         // float type
                double d = 3.1415;      //double type
                char c;                 //char type
                c = '\u0031';           //initial char type
                x = 12;                 //initial int type
         }
}
```

Write and Show Time

- Write 8 variables with 8 primitive types and use random numbers as their initial values.

  For example: `int i = 5;`

- Show it to your neighbor.

Assignment Compatibility

- In general, the value of one type cannot be stored in a variable of another type

```
int intVariable = 2.99; //Illegal
```

  - The above example results in a type mismatch because a **`double`** value cannot be stored in an **`int`** variable
- However, there are exceptions to this

```
double doubleVariable = 2;
```

  - For example, an **`int`** value can be stored in a **`double`** type

Assignment Compatibility

- More generally, a value of any type in the following list can be assigned to a variable of any type that appears to the right of it

  - Note that as your move down the list from left to right, the range of allowed values for the types becomes larger

Type casting

- An explicit ***type cast*** is required to assign a value of one type to a variable whose type appears to the left of it on the above list (e.g., **`double`** to **`int`**)

```
double aDoubleNum = 2.23;
int aInteger = (int)aDoubleNum;
```

- Note that in Java an **`int`** cannot be assigned to a variable of type **`boolean`**, nor can a **`boolean`** be assigned to a variable of type **`int`**

Operators

Category of operators:

- Arithmetic: `+ - * / % ++ --`
- Relational: `< > <= >= == !=`
- Logical: `! && ||`
- Conditional: `bool_expression ? true_case : false_case`
- Short-hand: `++ -- += -= *=` …
- Assignment: `=` (to store a value)
- Bitwise Operators: `& | ^ ~ >> << >>>` (unsigned shift right)
- Concatenate : `+`

Arithmetic Operators

- Addition/ Sum: `a + b`
- Subtraction/ Difference: `a - b`
- Multiplication/ Product: `a * b`
- Division/ Quotient: `a / b`
- Remainder (of integer division): `a % b`
- Negation/ Minus: `-a`

Arithmetic Operators and Expressions

- If an arithmetic operator is combined with **`int`** operands, then the resulting type is **`int`**
- If an arithmetic operator is combined with one or two **`double`** operands, then the resulting type is **`double`**
- If different types are combined in an expression, then the resulting type is the right-most type on the following list that is found within the expression

  - Exception: If the type produced should be **`byte`** or **`short`** (according to the rules above), then the type produced will actually be an **`int`**

Type in Expression

- Type promotion/conversion

  `byte` → `short` → `int` → `long` → `float` → `double`

```
byte   b = 104;
float  f = 3.14159f;
double d = 35 * f - 29.7 / b;
```

`[double ←int * float - double / byte]`

Integer and Floating-Point Division

- When one or both operands are a floating-point type, division results in a floating-point type

  `15.0 / 2` evaluates to `7.5`

- When both operands are integer types, division results in an integer type
  - Any fractional part is discarded
  - The number is not rounded

    `15 / 2` evaluates to `7`

- Be careful to make at least one of the operands a floating-point type if the fractional portion is needed

Discussion Time

Check the following codes carefully, and find the place where may cause compile error or overflow mistake:

```
public void method() {
        int i = 1, j;
        float f1 = 0.1;
        float f2 = 123;
        long l1 = 12345678, l2 = 8888888888;
        double d1 = 2e20, d2 = 124;
        byte b1 = 1, b2 = 2, b3 = 129;
        j = j + 10;
        i = i / 10;
        i = i * 0.1;
        char c1 = 'a', c2 = 125;
        byte b = b1 - b2;
        char c = c1 + c2 - 1;
        float f3 = f1 + f2;
        float f4 = f1 + f2 * 0.1;
        double d = d1 * i + j;
        float f = (float)(d1 * 5 + d2);
}
```

The % Operator -- remainder

- The `%` operator is used with operands of type `int` to recover the information lost after performing integer division
  - `15 / 2` evaluates to the quotient `7`
  - `15 % 2` evaluates to the remainder `1`

String Concatenation

- "**+**" can be used to concatenate two string
  - `int id = 80 + 90;` // addition
  - `String s = "hello" + "world";` // concatenation
- If one of the operands is a String type, the other one will be converted to String automatically.
- When using `System.out.println(??)`, the content `??` will be printed as `String` type.
  - What is the result for "System.*out.println(1+2);*"**?**
  - What is the result for "System.*out.println("The result is:"+1+2);*"

Shorthand Assignment Statements

Example:	Equivalent To:
`count += 2;`	`count = count + 2;`
`sum -= discount;`	`sum = sum - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time / rushFactor;`
`change %= 100;`	`change = change % 100;`

Shorthand Assignment Statements

Operation	Example	Equivalent Expression
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b

Conditional Operators

- `x ? y : z`
- `x` is a `boolean` expression. If `x` is `true`, then return the value of `y`, else return the value of `z`
- E.g.:

```
int score = 80;
String type = score < 60 ? "failed" : "pass";
System.out.println("type = " + type);
```

Increment(++) and Decrement (--)Operators

```
public class Test {
   public static void main(String[] args) {
        int x = 10, y = 20;
        int i = y++;
        System.out.print("i = " + i);
        System.out.println(" y = " + y);
        i = ++y;
        System.out.print("i = " + i);
        System.out.println(" y = " + y);
        i = --x;
        System.out.print("i = " + i);
        System.out.println(" x = " + x);
        i = x--;
        System.out.print("i = " + i);
        System.out.println(" x = " + x);
   }
}
```

Output:

```
i = 20  y = 21
i = 22  y = 22
i = 9  x = 9
i = 9  x = 8
```

Note:

- ++(--)
- When before variable, calculation first;
- When after variable, assign the value first;

Logical Operators

- Relational operators
  - < > <= >= == !=
  - result is a **`boolean`** value

e.g.

the value of 3 == 7 is false

the value of 3 != 7 is true

the value of 3 > 7 is false

the value of 3 <= 7 is true

Logical Operators

- Boolean operators: !, &&, ||
  - operate on **`boolean`** values
  - result is a **`boolean`** value
  - **`&&`** (logical and), **`||`** (logical or) are short-circuiting, i.e., the second argument is not evaluated if the first argument already determines the value.

e.g.

$expression_1$ && $expression_2$

3 == 7 && 3 <= 7

3 == 7 || 3 <= 7

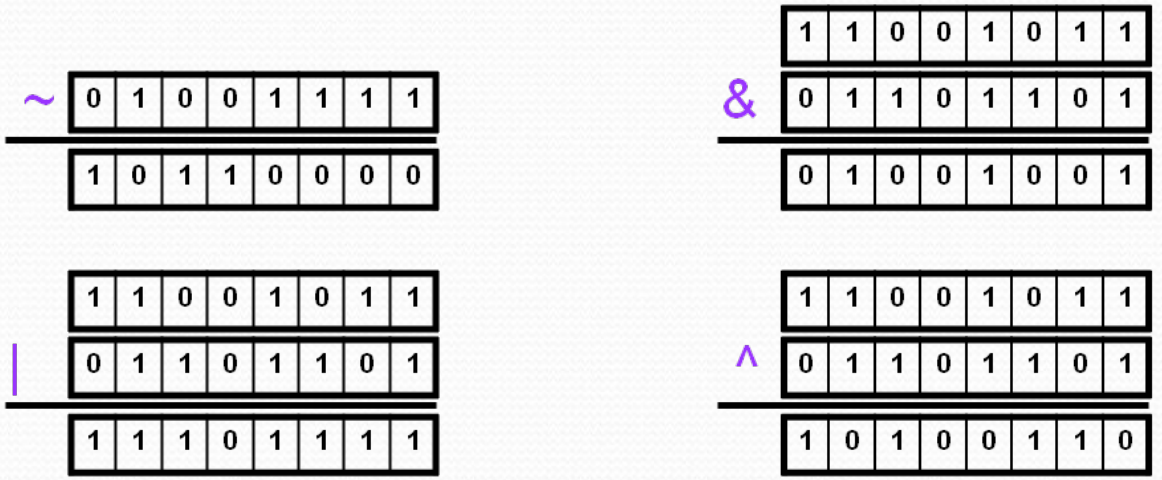
```
x != 0 && 1 / x > x + y // no division by 0
```

the second part is never evaluated if equals zero.

a	b	!a	a&&b	a||b
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

Bitwise Operators

- Bitwise operators
  - Work with any of the integer types
  - Work directly on the bits that make up the integers
  - **&** (“and”), **|** (“or”), **^** (“xor”), **~** (“not”)



Bitwise Operators

- Bitwise operators
  - There are also > > and < < operators, which shift a bit pattern to the right or left. These operators are often convenient when you need to build up bit patterns to do bit masking:

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

  - a >>> operator fills the top bits with zero, whereas > > extends the sign bit into the top bits. There is no <<< operator

Mathematical Functions and Constants

- The Math class contains an assortment of mathematical functions

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // prints 2.0
```

```
double y = Math.pow(x, a);
Math.sin
Math.cos
Math.tan
Math.atan
Math.exp
Math.log
Math.PI
Math.E
```

you can avoid the Math prefix for the mathematical methods and constants by adding the following line to the top of your source file:

```
import static java.lang.Math.*;
```

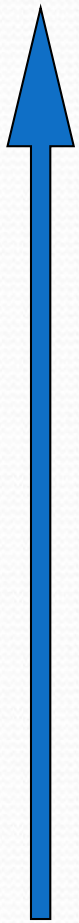
Parentheses and Precedence Rules

- An expression can be *fully parenthesized* in order to specify exactly what sub expressions are combined with each operator
- If some or all of the parentheses in an expression are omitted, Java will follow *precedence* rules to determine, in effect, where to place them

Always include parentheses in practice!

Operator Precedence Rules

high



low

Separator	. ( ) { } ; ,
Associative	Operators
R to L	++ -- ~ ! (data type)
L to R	* / %
L to R	+ -
L to R	<< >> >>>
L to R	< > <= >= instanceof
L to R	== !=
L to R	&
L to R	^
L to R	|
L to R	&&
L to R	||
R to L	?:
R to L	= *= /= %= += -= <<= >>= >>>= &= ^= |=

Operator Precedence Rules

- Examples:
  - a && b || c means (a && b) || c
  - a += b += c means a += (b += c)

Always include parentheses in practice!

Importing Packages and Classes

- Code libraries in Java are called ***packages.***
  - A package is a **collection of classes** that is stored in a manner that makes it easily accessible to any program.
  - In order to use a class that belongs to a package, the class must be brought into a program using an **`import`** statement.
  - Classes found in the package **`java.lang`** are imported automatically into every Java program.

```
import java.text.NumberFormat;
  // import theNumberFormat class only
import java.text.*;
  //import all the classes in package java.text
```

- Let's check Java's Math API!

Example: java.lang.Math

```
// java.lang.Math is imported automatically.
public class Test {
   public static void main(String[] args) {
      double i = -3.24;
      double j = 56.2;
      System.out.println(Math.abs(i));
      System.out.println(Math.max(i, j));
      System.out.println(Math.sqrt(j));
   }
}
```

Flow of Control

- **`if-else`**, **`if`**, and **`switch`** statements.
- **`while`**, **`do-while`**, and **`for`** statements.
- A Boolean expression evaluates to either **`true`** or **`false`** -- **used to control the flow**

`if-else` Statement

- An **`if-else`** statement chooses between two alternative statements based on the value of a Boolean expression:

```
if(Boolean_Expression)
   Yes_Statement
else
   No_Statement
```

- Each ***`Yes_Statement`*** and ***`No_Statement`*** branch of an **`if-else`** can be a made up of a single statement or a **compound statement.**

Compound Statements

Compound Statement: a statement that is made up of a list of statements and enclosed in a pair of curly brackets({ }).

```
if(myScore > yourScore) {
    System.out.println("I win!");
    wager = wager + 100;
} else {
    System.out.println("So sad...");
    wager = 0;
}
```

Multiway `if-else` Statement

```
if(Boolean_Expression)
    Statement_1
else if(Boolean_Expression)
    Statement_2
        :
else if(Boolean_Expression_n)
    Statement_n
else
  Statement_For_All_Other_Possibilities
```

Example: if-else

```
public class Test {
    public static void main(String[] args) {
        int i = 20;
        if(i < 20) {
                System.out.println("<20");
        } else if(i < 40) {
                System.out.println("<40");
        } else if(i < 60) {
                System.out.println("<60");
        } else {
                System.out.println(">=60");
        }
    }
}
```

The `switch` Statement

- The **`switch`** statement is the only other kind of Java statement that implements **multiway** branching.
  - When a **`switch`** statement is evaluated, one of a number of different branches is executed.
  - The choice of which branch to execute is determined by a **controlling expression** enclosed in parentheses after the keyword **`switch`** .
    - Before Java SE 7, the controlling expression must evaluate to a **`char`**, **`int`**, **`short`**, or **`byte`**.
    - Start from Java SE 7, switch supports **`String`** type in the switch statement.

The `switch` Statement

```
switch(expression){
    case value :
       //statement
       break; // optional
    case value :
       //statement
       break; // optional
              …
     //You can have any number of case statements
              …
    default : // optional
       //statement
}
```

- Execution starts at the case label that matches the value on which the selection is performed and continues until the next break or the end of the switch.
- If none of the case labels match, then the default clause is executed, if it is present.

The `switch` Statement

```
switch(numberOfFlavors) {
  case 32:
       System.out.println("So many yummy flavors!! ");
       break;
  case 1:
       System.out.println("I bet it is vanilla");
       break;
  case 2:
  case 3:
  case 4:
        System.out.println(numberOfFlavors + " is ok");
        break;
  default:
       System.out.println("How many do you have?");
       break;
}
```

- Output result if **`numberOfFlavors`** equals to 32, 3, 5?

The `switch` Statement

```
switch(numberOfFlavors) {
  case 32:
       System.out.println("So many yummy flavors!! ");
       break;
  case 1:
       System.out.println("I bet it is vanilla");
       break;
  case 2:
  case 3:
  case 4:
        System.out.println(numberOfFlavors + " is ok");
        break;
  default:
       System.out.println("How many do you have?");
       break;
}
```

- Output result if **`numberOfFlavors`** equals to 32, 3, 5?

Example: switch

```
public class Test {
   public static void main(String args[]){
      char grade = 'C';
      switch(grade)
      {
         case 'A' :
            System.out.println("Excellent");
            break;
         case 'B' :
         case 'C' :
            System.out.println("Good");
            break;
         case 'D' :
            System.out.println("Pass");
         case 'F' :
            System.out.println("Work harder");
            break;
         default :
            System.out.println("Unknown");
      }
      System.out.println("Your grade is " + grade);
   }
}
```

```
Good
Your grade is C
```

Example: switch

```
String color = “blue”;
switch (color) {
        case "blue":
                System.out.println("BLUE");
                break;
        case "red":
                System.out.println("RED");
                break;
        default:
                System.out.println("INVALID COLOR CODE");
}
```

```
BLUE
```

Boolean Expressions

- A **`boolean`** expression is an expression that is either **`true`** or **`false`**.
- The simplest **`boolean`** expressions compare the value of two expressions:

```
time < limit
yourScore == myScore
```

  - Note that Java uses two equal signs (**`==`**) to perform equality testing: a single equal sign (**`=`**) is used only for assignment.
  - Use parentheses in practice, although a **`boolean`** expression does not have to be enclosed in parentheses.

Java Comparison Operators

Display 3.3 Java Comparison Operators

MATH NOTATION	NAME	JAVA NOTATION	JAVA EXAMPLES
=	Equal to	==	`x + 7 == 2*y` `answer == 'y'`
≠	Not equal to	!=	`score != 0` `answer != 'y'`
>	Greater than	>	`time > limit`
≥	Greater than or equal to	>=	`age >= 21`
<	Less than	<	`pressure < max`
≤	Less than or equal to	<=	`time <= limit`

Pitfall: Using == with Float point

- When comparing the equality of two float point numbers, we usually do not use == for this purpose.
- We usually compare their difference to a very small number.

```
float a = 0.1f;
float b = 0.1f;
for (int i = 1; i < 11; i++) {
        a += .1;
    }
b *= 11;
if (a!=b)
    System.out.println("a does not equal to b");

float c = Math.abs(b - a);
if(c < 1e-6) {
    System.out.println("a equals to b");
}
```

Example: ==

```
public class Test {
    public static void main(String[] args) {
        float a = 3.141234567f;
        double b = 3.141234567;
        //float b = 3.141234567f;
        if(a == b) {
            System.out.println("a is equal to b");
        } else {
            System.out.println("a is not equal to b");
        }
        if((a - b) < 1e-4) {
            System.out.println("Equal!");
        }
    }
}
```

```
a is not equal to b
Equal!
```

Building Boolean Expressions

- Conjuction: **p and q** Java we write: `p && q`
- Disjunction: **p or q** Java we write: `p || q`
- Negation: **not p** Java we write: `!p`

- Multiple inequalities must be joined by `&&`
  - Use `(min < result) && (result < max)` to represent `min < result < max`

- Boolean expressions evaluate to `true` or `false`

```
boolean madeIt = (time < limit) && (limit < max);
```

Loops

- **Loops** in Java are similar to those in other high-level languages (like C).
- Java has three types of loop statements: the **`while`**, the **`do-while`**, and the **`for`** statements.
  - The code that is repeated in a loop is called the **body** of the loop.
  - Each repetition of the loop body is called an **iteration** of the loop.

`while` statement

- A **`while`** statement is used to repeat a portion of code (i.e., the loop body) based on the evaluation of a boolean expression.
- The boolean expression is checked **before** the loop body is executed.

`while` Syntax

```
while(Boolean_Expression)
    Statement
```

Or

```
while(Boolean_Expression)  {
    Statement_1
    Statement_2

    Statement_Last
}
```

do-while Statement

- A **do-while** statement is similar to a while statement, with the exception that the boolean expression is checked **after** the loop body is executed.

```
do
    Statement
while(Boolean_Expression);
```

Or

```
do {
    Statement_1
    Statement_2
        .
        .
        .
    Statement_Last
} while(Boolean_Expression);
```

Example: while, do-while

```
public class Test {
	public static void main(String[] args) {
		int i = 0;
		while(i < 10) {
			System.out.println(i);
			i++;
		}
		i = 0;
		do {
			i++;
			System.out.println(i);
		} while(i < 10);
	}
}
```

The `for` Statement

- The `for` statement is most commonly used to step through an integer variable in equal increments.
- It begins with the keyword `for`, followed by three expressions in parentheses that describe what to do with one or more **controlling variables**.
  - The first expression tells how the control variable or variables are **initialized** before the first iteration.
  - The second expression determines **when the loop should continue**, based on the evaluation of a boolean expression **before** each iteration.
  - The third expression tells how the control variable or **variables** are **updated** after each iteration of the loop body.

`for` Statement Syntax

Display 3.10 **for Statement Syntax and Alternate Semantics *(Part 1 of 2)***

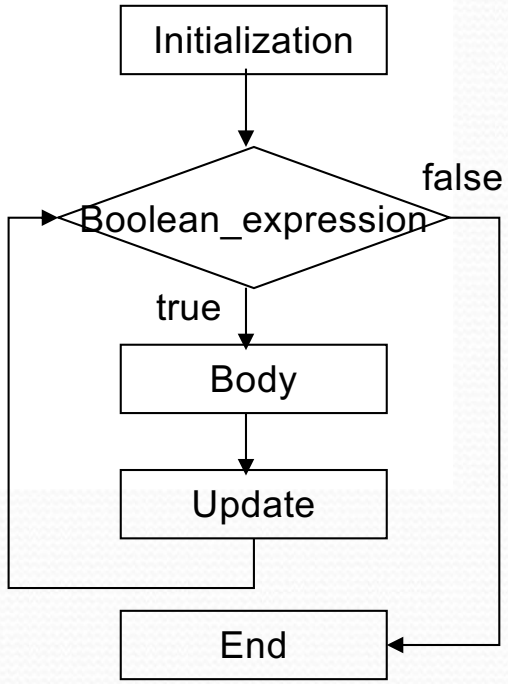
for STATEMENT SYNTAX:

SYNTAX:

```
for (Initialization; Boolean_Expression; Update)
    Body
```

EXAMPLE:

```
for (number = 100; number >= 0; number--)
    System.out.println(number
        + " bottles of beer on the shelf.");
```



Example: for loop

```
public class Test {
	public static void main(String[] args) {
		long result = 0;
		long f = 1;
		for(int i = 1; i <= 10; i++) {
			f = f * i;
			result += f;
		}
		// result: 1! + 2! + ... + 10!
		System.out.println("result=" + result);
	}
}
```

Write and Show Time

Write a while loop equivalent to this for loop:

```
for(int number = 100; number >= 1; number--) {
   System.out.println(number + " bottles of beer left!");
}
```

Infinite Loops

- A **`while`**, **`do-while`**, or **`for`** loop should be designed so that the value tested in the boolean expression is changed in a way that eventually makes it false, and terminates the loop.
- If the boolean expression remains true, then the loop will run forever, resulting in an **infinite loop**.

Note: loops that check for numerical equality or inequality (`==` or `!=`) are especially prone to this error and should be avoided if possible (floating point equality is difficult to determine!)

The `break` and `continue` Statements

- The **`break`** statement: **`break;`**
  - When executed, the **`break`** statement ends the nearest enclosing switch or loop statement.
- The **`continue`** statement: **`continue;`**
  - When executed, the **`continue`** statement ends the current loop body iteration of the nearest enclosing loop statement.
  - Note that in a **`for`** loop, the **`continue`** statement transfers control to the **update** expression.

The **break** and **continue** Statements

```
for(int i = 0; i < 100; i++) {
    if(i == 47)
        break; // Exit out of for loop
    if(i % 9 != 0)
        continue; // Go to next iteration immediately
    System.out.println(i);
}
int j = 0;
while(true) {
    j++;
    if(j == 47)
        break; // Exit out of loop
    if(j % 10 != 0)
        continue; // Go to top of loop immediately
    System.out.println(j);
}
```

Example: break

```
public class Test {
   public static void main(String[] args) {
      int stop = 4;
      for(int i = 1; i <= 6; i++) {
         if(i == stop)
            break;
         System.out.println("i = " + i);
      }
   }
}
```

Example: continue

```
public class Test {
   public static void main(String[] args) {
      int skip = 4;
      for(int i = 1; i <= 6; i++) {
         if(i == skip)
            continue;
         System.out.println("i = " + i);
      }
   }
}
```

General Debugging Techniques

- Examine the system as a whole – don't assume the bug occurs in one particular place.
- Try different test cases and check the input values.
- Comment out blocks of code to narrow down the offending code.
- Check common pitfalls.
- Take a break and come back later.
- DO NOT make random changes just hoping that the change will fix the problem!

Tips for Productive Coding

- **Plan Globally**
  - Design the overall function of the program first.
- **Develop Incrementally**
  - Write a little bit of code at a time and test it before moving on.

Write and Show Time - MultiTable

```
1*1=1
1*2=2   2*2=4
1*3=3   2*3=6   3*3=9
1*4=4   2*4=8   3*4=12  4*4=16
1*5=5   2*5=10  3*5=15  4*5=20  5*5=25
1*6=6   2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7   2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8   2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9   2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
```

Write and Show Time

```
public class Test {
    public static void main(String[] args) {
        for(int i = 1; i < 10; i++) {      //loop row
            for(int j = 1; j <= i; j++) { //loop column
                System.out.print(j + "*" + i + "=" + i*j);
                if(i * j < 10) // only one digit of output
                    System.out.print("   ");
                else
                    System.out.print("  ");
            }
            System.out.println();
        }
    }
}
```

Summary

- Keywords
- Variable Declarations and Assignments
- Standard input and output (I/O)
- Data Types
- Type Casting
- Arithmetic Expressions and Operators
- Java Packages
- Flow Control