AP Computer Science A

Java Programming Essentials [Ver.4.0]

Unit 1: Using Objects and Methods

CHAPTER 3: OPERATORS AND EXPRESSIONS

DR. ERIC CHOU
IEEE SENIOR MEMBER

AP Computer Science Curriculum

- Casting and Range of Variables (T 1.5)
- Compound Assignment Operators (T 1.6)

Objectives:

- Program Units
- Static Programs
- Object-Oriented Programs
- Operators
- Increments/Decrements
- Integer Division and Modulus Arithmetic
- Operator Precedence
- Implicit and Explicit Casting.
- Determination of Data Type for Operation Results.

Overview

Lecture 1

Objectives

Elementary Programming

- Program Units – Variable, Expression, Statement, Function, Class, Program
- Static Programming (Procedural Programming)
- Object-Oriented Programming

Objectives

Operators and Expressions

- Operators
- Expressions
- Operator Precedence

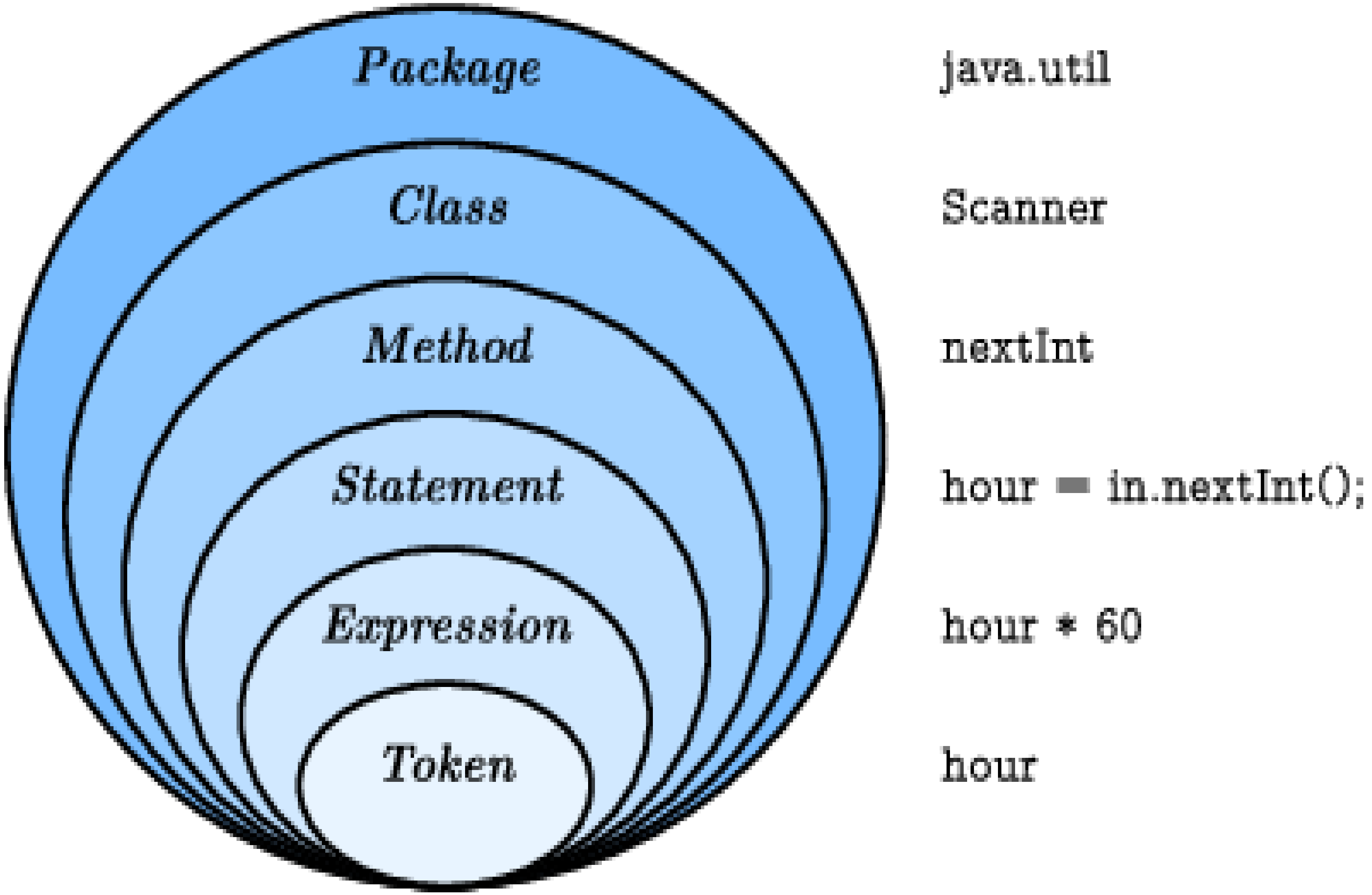
Objectives

Computation

- Problem and Problem Solving
- Algorithm
- Pseudo Code
- Coding and Debugging

Program Units

Lecture 2


Package
java.util
Class
Scanner
Method
nextInt
Statement
hour = in.nextInt();
Expression
hour * 60
Token
hour

Expressions

- An **expression** is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value. You've already seen examples of expressions, illustrated in bold below:

  int **cadence = 0**;
  anArray[0] = 100;
  System.out.println(**"Element 1 at index 0: " + anArray[0]**);
  int **result = 1 + 2**; // result is now 3
  if (**value1 == value2**)
      System.out.println(**"value1 == value2"**);

Statements

- Statements are roughly equivalent to sentences in natural languages. A **statement** forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).
  - Assignment expressions
  - Any use of ++ or --
  - Method invocations
  - Object creation expressions

Statements

- Such statements are called **expression** *statements*. Here are some examples of expression statements.

  Assignment Statement :

  ```
  aValue = 8933.234;
  ```

  Increment Statement:

  ```
  aValue++;
  ```

  Method Invocation Statement :

  ```
  System.out.println("Hello World!");
  ```

  Object Creation Statement :

  ```
  Bicycle myBike = new Bicycle();
  ```

Statements

- In addition to expression statements, there are two other kinds of statements: **declaration** *statements* and **control flow** *statements*.
- A **declaration statement** declares a variable. You've seen many examples of declaration statements already:
  // declaration statement double aValue = 8933.234;

Finally, **control flow** *statements* regulate the order in which statements get executed. You'll learn about control flow statements in the next section, Control Flow Statements

Control Flow Statements

- The statements inside your source files are generally executed from top to bottom, in the order that they appear.
- **Control flow statements**, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to **conditionally** execute particular blocks of code.
- This section describes
    - the decision-making statements (`if-then`, `if-then-else`, `switch`),
    - the looping statements (`for`, `while`, `do-while`), and
    - the branching statements (`break`, `continue`, `return`)

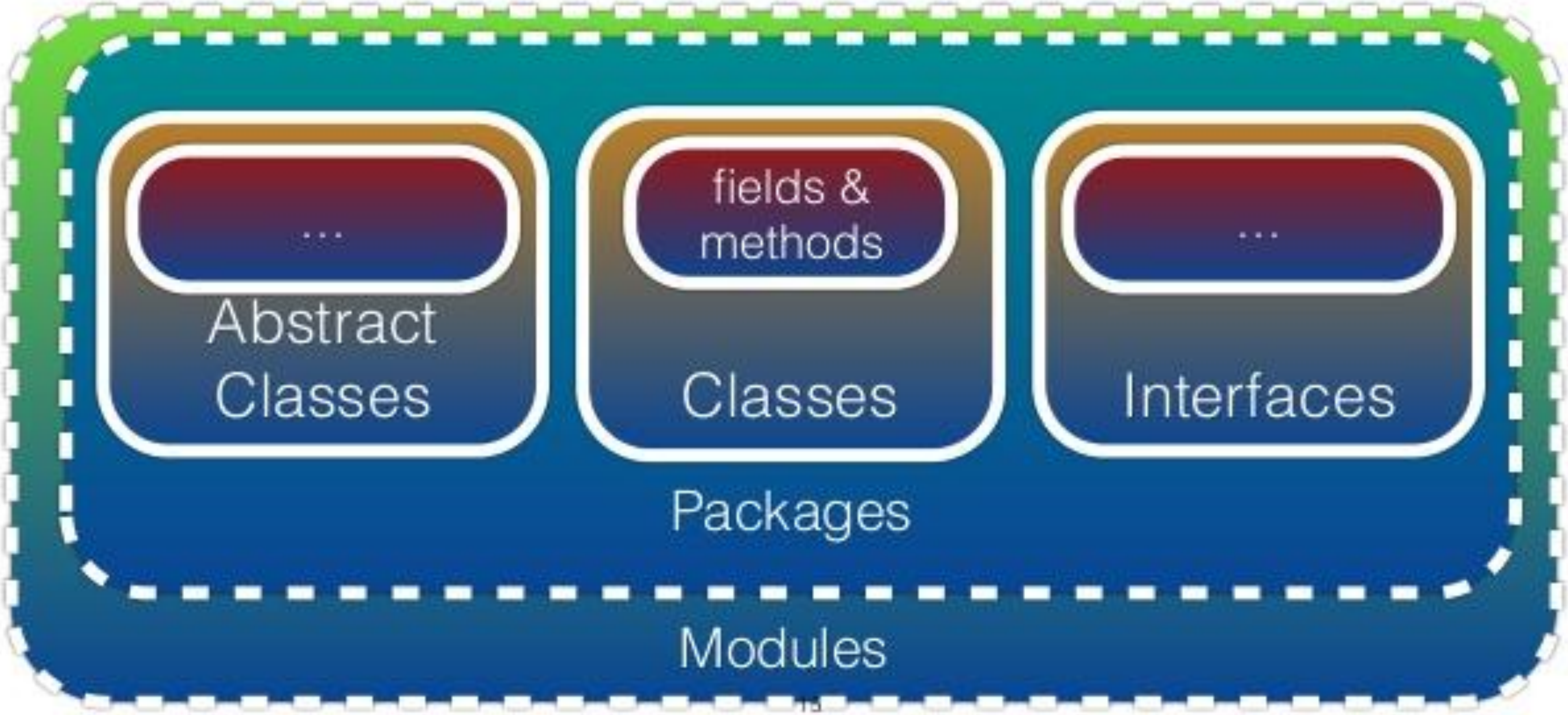
  supported by the Java programming language.

Blocks

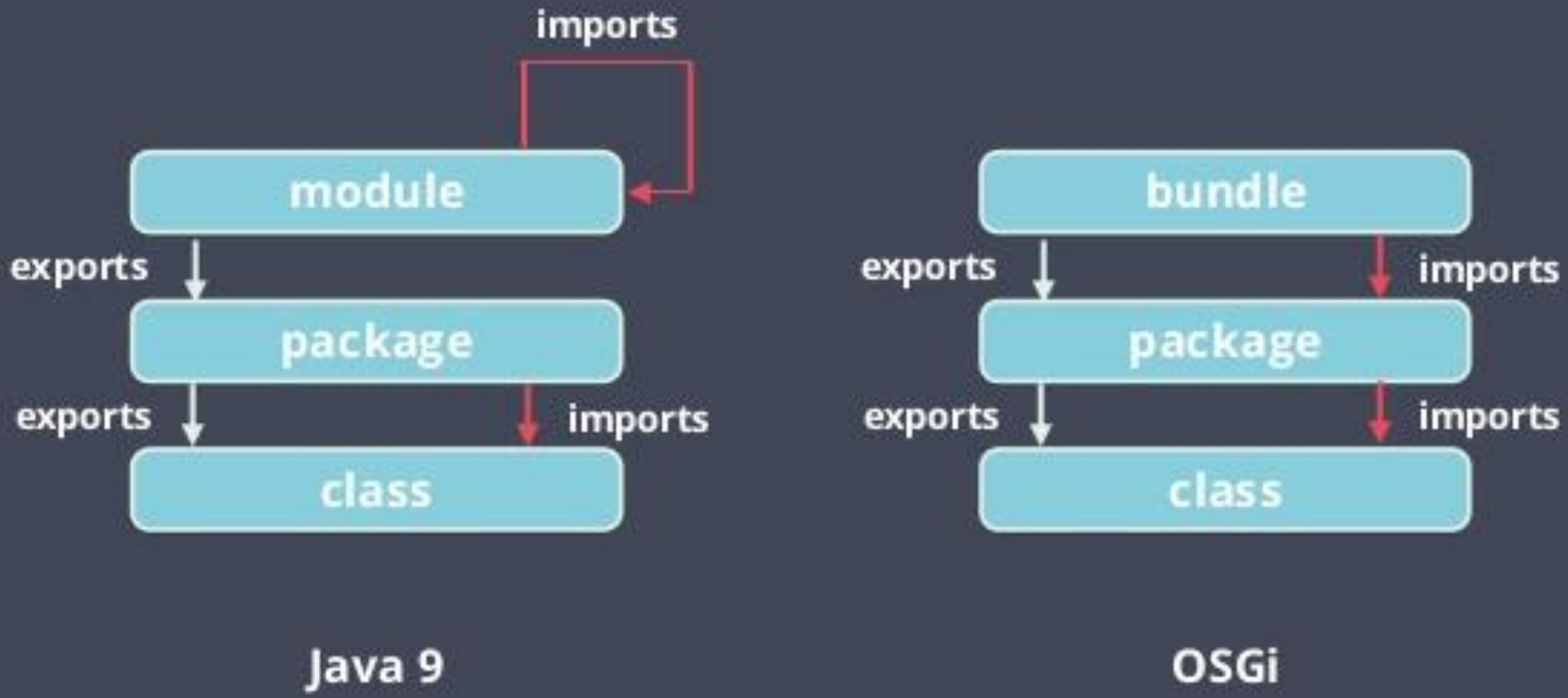
- A **block** is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, BlockDemo, illustrates the use of blocks:

```
class BlockDemo {
   public static void main(String[] args) {
      boolean condition = true;
      if (condition) { // begin block 1
        System.out.println("Condition is true.");
      } // end block one
      else { // begin block 2
        System.out.println("Condition is false.");
      } // end block 2
   }
}
```

Code encapsulation

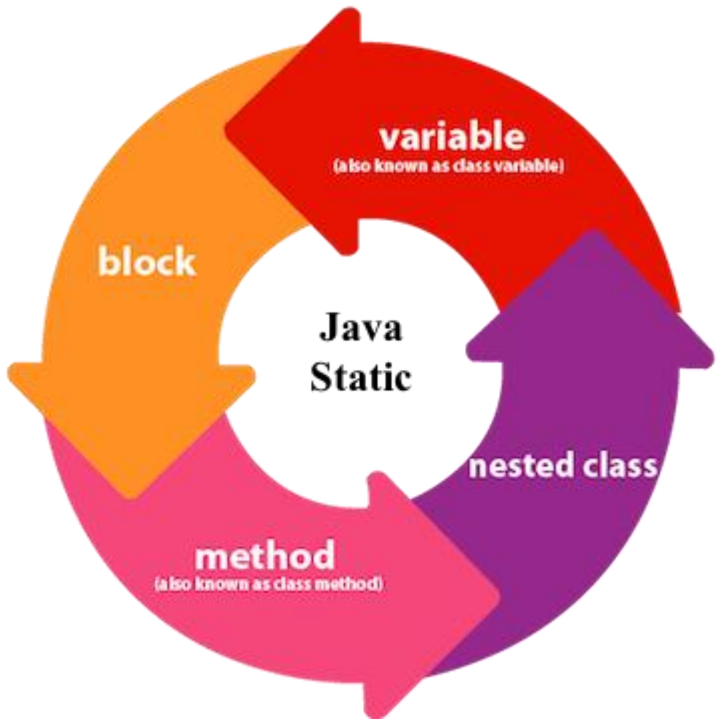


Module vs package dependencies



Static Program

Lecture 3



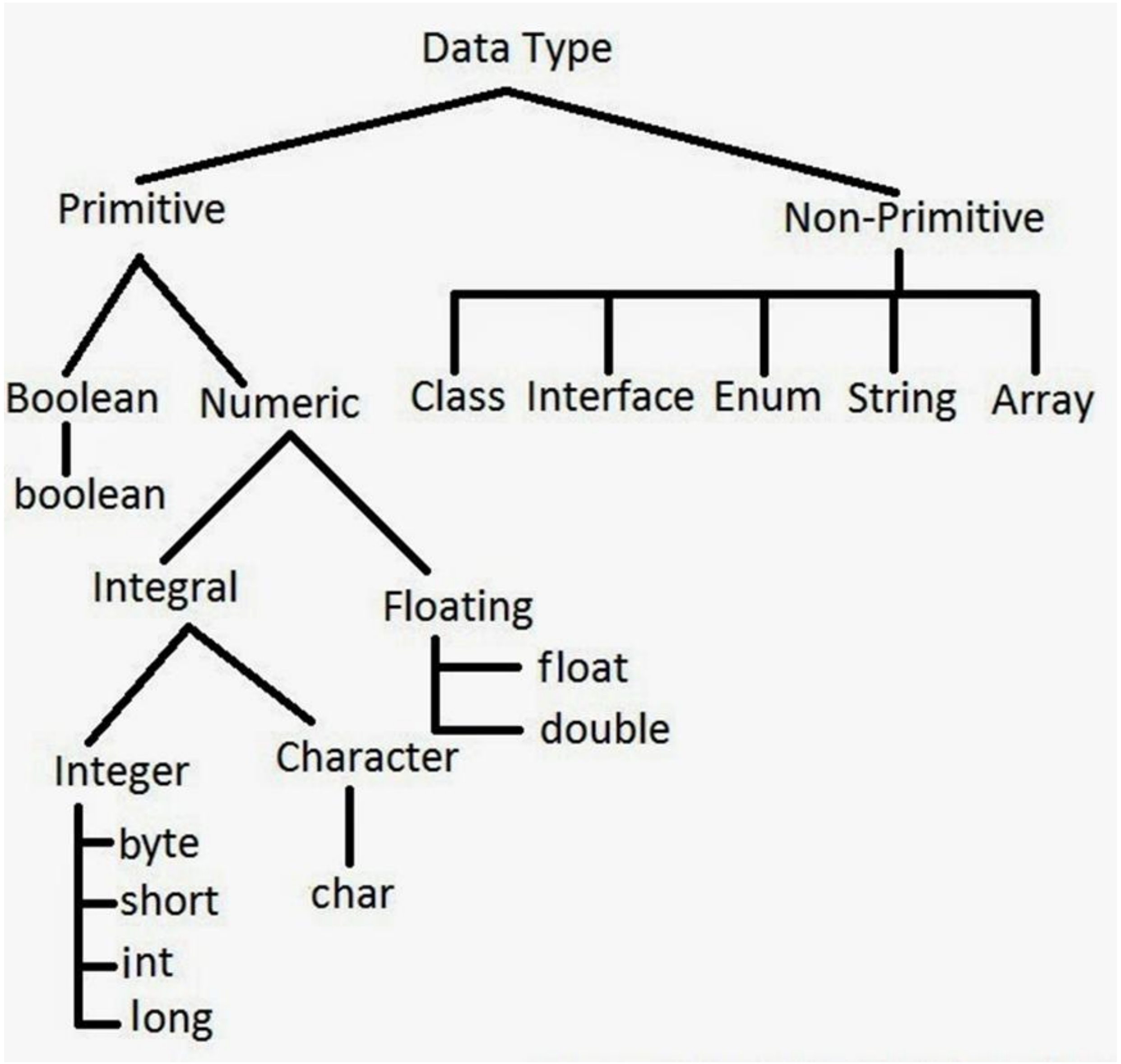
```
public class Physics {
   public static double LIGHT_SPEED = 3E8;

   public static double energy(double mass){
      return mass * LIGHT_SPEED * LIGHT_SPEED;
   }

   public static void main(String[] args){
      double mass = 1.0;
      double energyOfMass = energy(mass);
      System.out.println(energyOfMass);
   }
}
```

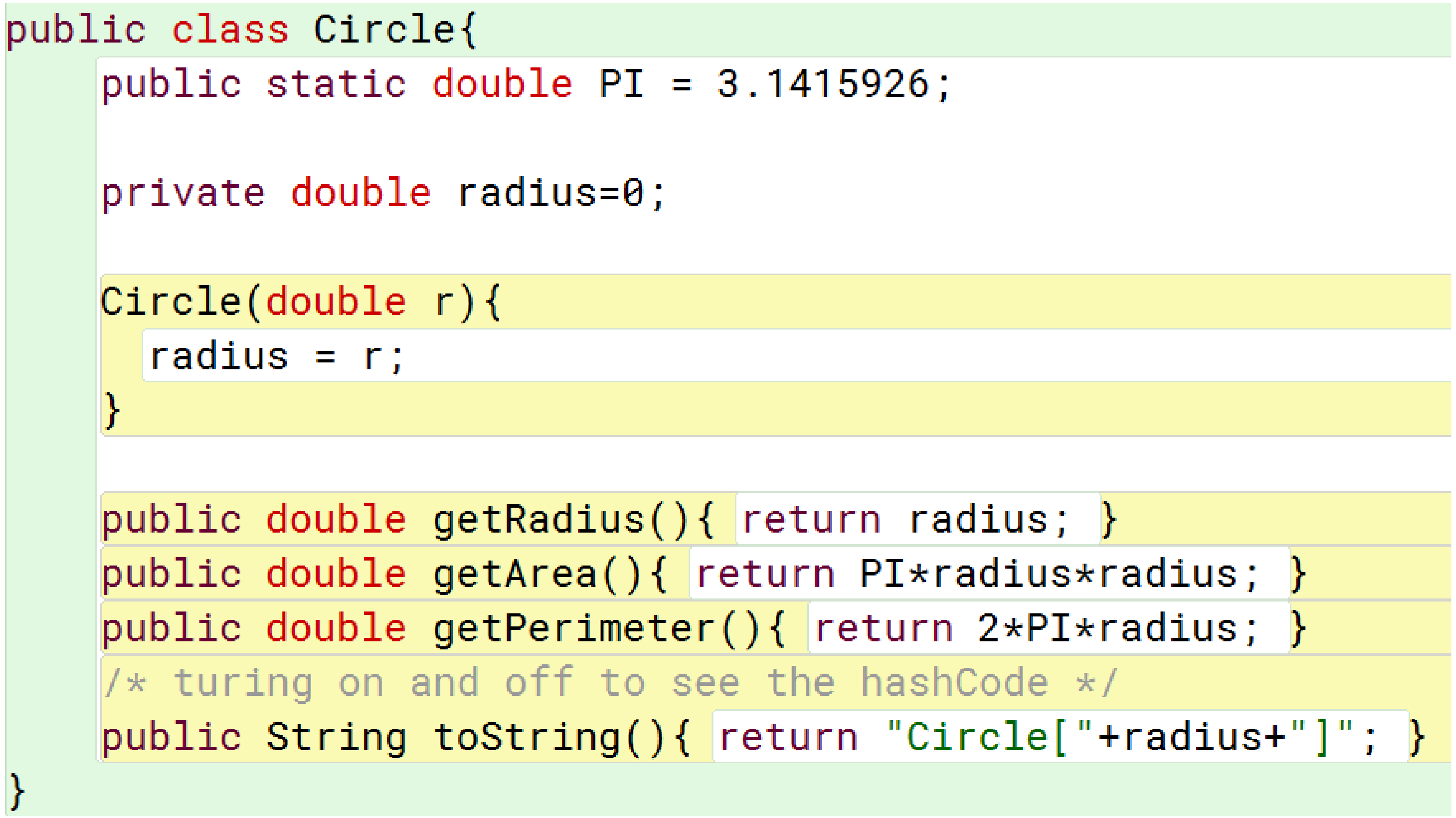
Object-Oriented Program

Lecture 4

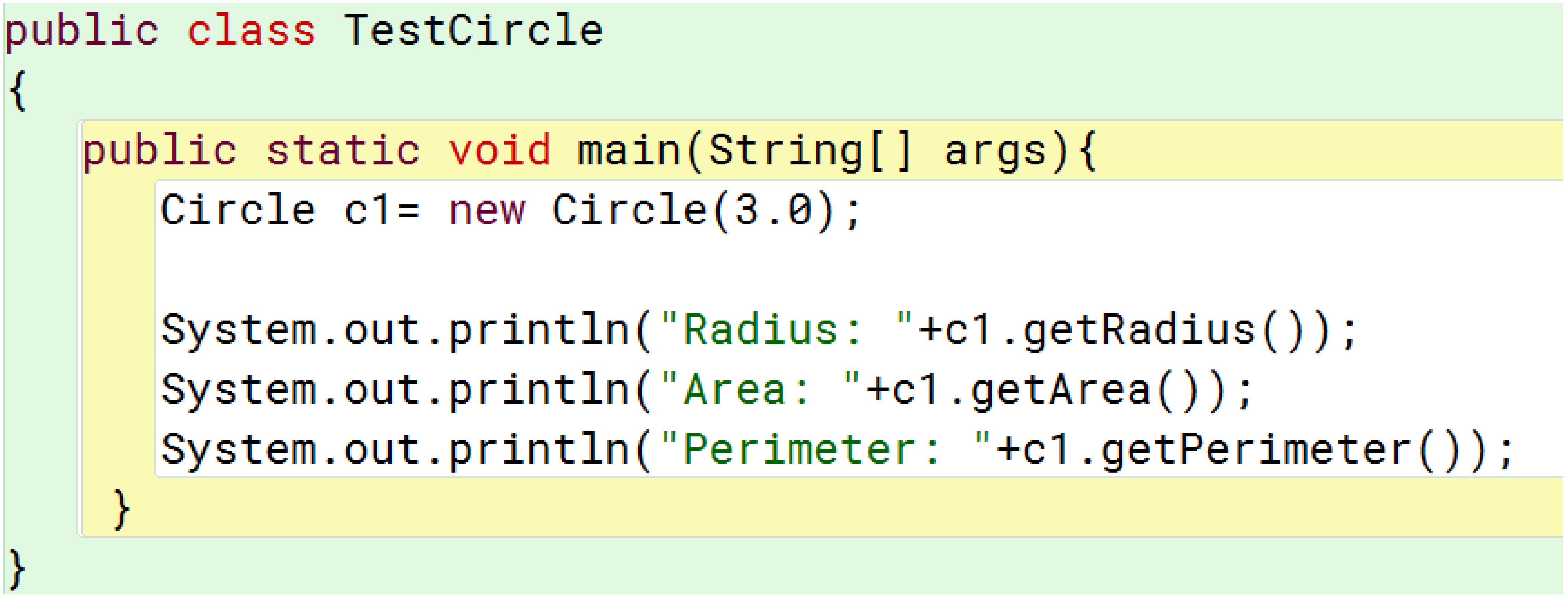
Class

- A class is a module (Program unit) and a template for object (data record type). It can serve for two purposes. (When compared to C++ language).
- Class, String and Array are three reference data type in Java language.
- Reference Data Type is used for data types that don’t have fixed memory size. And the data entity exists only at run-time in Heap memory area.

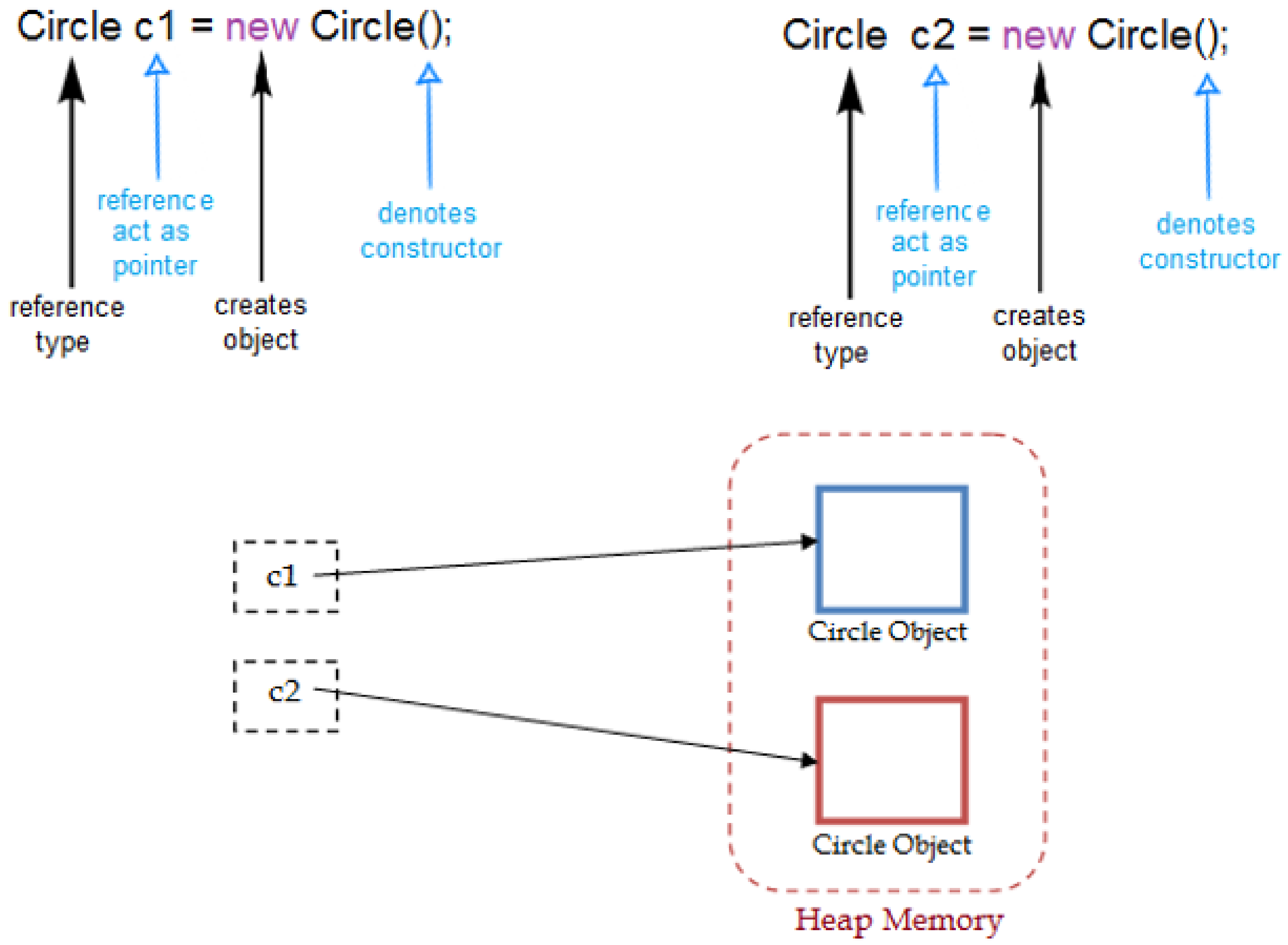

Data Type
Primitive
Non-Primitive
Boolean
Numeric
boolean
Integral
Floating
float
double
Integer
Character
byte
short
int
long
char
Class
Interface
Enum
String
Array



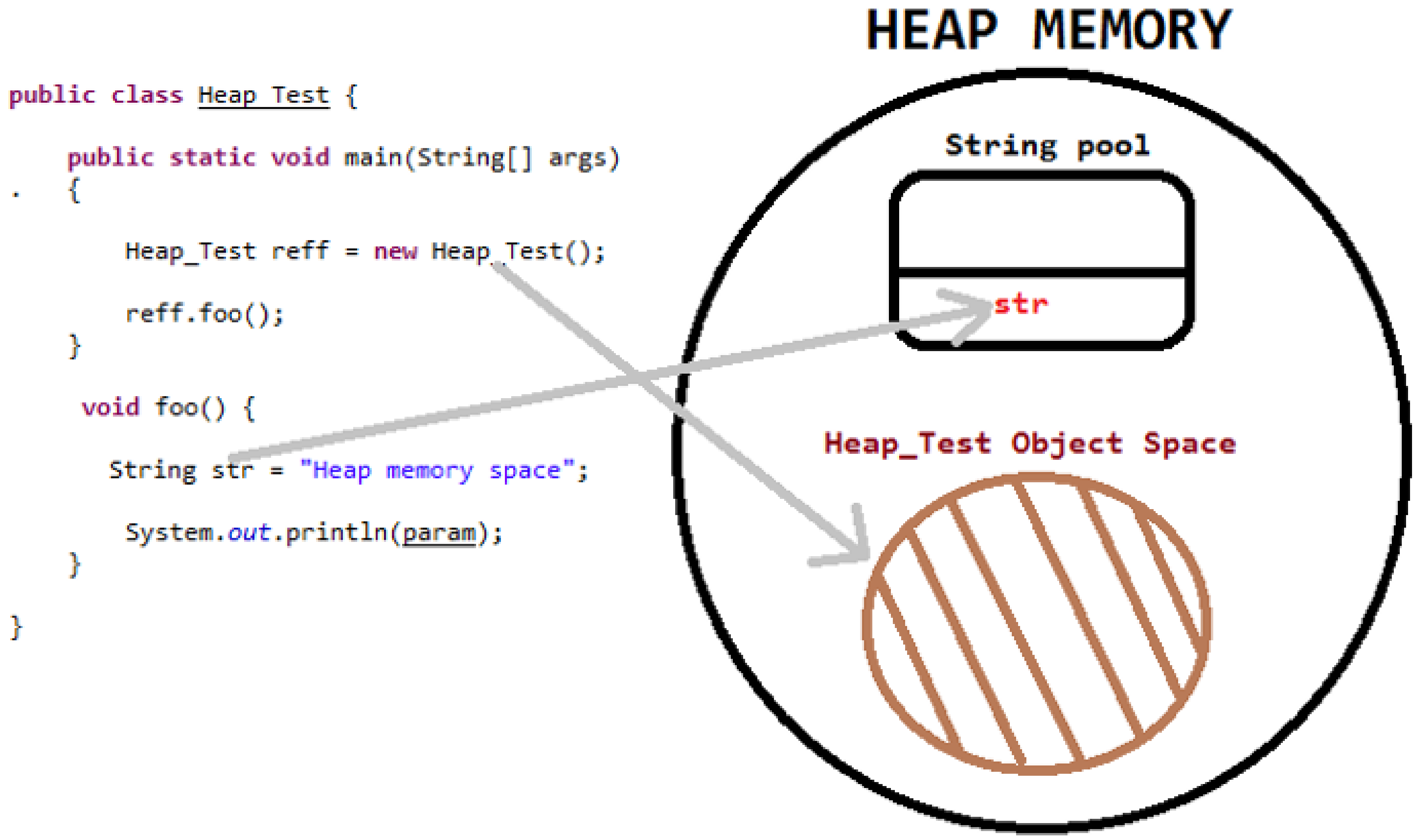
```
public class Circle{
    public static double PI = 3.1415926;

    private double radius=0;

    Circle(double r){
      radius = r;
    }

    public double getRadius(){ return radius; }
    public double getArea(){ return PI*radius*radius; }
    public double getPerimeter(){ return 2*PI*radius; }
    /* turing on and off to see the hashCode */
    public String toString(){ return "Circle["+radius+"]"; }
}
```



```java
public class TestCircle
{
    public static void main(String[] args){
        Circle c1= new Circle(3.0);

        System.out.println("Radius: "+c1.getRadius());
        System.out.println("Area: "+c1.getArea());
        System.out.println("Perimeter: "+c1.getPerimeter());
    }
}
```



```
Circle c1 = new Circle();
Circle  c2 = new Circle();
```



```java
public class Heap_Test {

    public static void main(String[] args)
.   {

        Heap_Test reff = new Heap_Test();

        reff.foo();
    }

     void foo() {

       String str = "Heap memory space";

        System.out.println(param);
    }

}
```

In-Class Demonstration Program

- Constant Pi
- Constructor
- Getter method
- Setter method
- getArea()
- getPerimeter()

Circle.java and TestCircle.java

Operators

Lecture 5

Arithmetic Operators

- The Java programming language supports various arithmetic operators for all floating-point and integer numbers.
- These operators are + (addition), - (subtraction), * (multiplication), / (division), and % (modulo).
- The following table summarizes the binary arithmetic operations in the Java programming language.

Arithmetic Operators

Operator	Use	Description
+	op1 + op2	Adds op1 and op2; also used to concatenate strings
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2

Demonstration Program

ArithmeticDemo.java

Data Type of the Result

- Note that when an integer and a floating-point number are used as operands to a single arithmetic operation, the result is **floating point**.
- The integer is implicitly converted to a floating-point number before the operation takes place.
- The following table summarizes the data type returned by the arithmetic operators, based on the data type of the operands. The necessary conversions take place before the operation is performed.

Data Type of the Result

Data Type of Result	Data Type of Operands
long	Neither operand is a float or a double (integer arithmetic); at least one operand is a long.
int	Neither operand is a float or a double (integer arithmetic); neither operand is a long.
double	At least one operand is a double.
float	At least one operand is a float; neither operand is a double.

Promotion/Negation of Data Type

- In addition to the binary forms of + and -, each of these operators has unary versions that perform the following operations, as shown in the next table:

Operator	Use	Description
+	+op	Promotes op to int if it's a byte, short, or char
-	-op	Arithmetically negates op

Integer Division

- **/** is the division operator. If the types of the operands are **double**, then "real" division is performed. Real division is normal division as you learned in grade school. Division is approximate on a computer because you can't do infinitely precise division (recall that **double** values have finite precision, so there is usually a tiny, but unavoidable error representing real numbers). The result of dividing a **double** by a **double** is a **double**. However, if both expressions have type **int**, then the result is an **int**. Does that sound strange? What is the result of evaluating **7 / 3**?
- Java does integer division, which basically is the same as regular real division, but you throw away the remainder (or fraction). Thus, **7 / 3** is **2** with a remainder of **1**. Throw away the remainder, and the result is 2.

Integer Division

- What happens if you really want an answer that's **double**. That is, you want the answer to be **2.33333333** or something close.
- One way is to **cast** the denominator to **double**. You do this as **7 / (double) 3**. Casting is an operator that creates a temporary **double** value. Thus it creates a temporary **3.0** double.
- When one of the operands to a division is a **double** and the other is an **int**, Java implicitly (i.e. behind your back) casts the **int** operand to a **double**. Thus, Java performs the real division **7.0 / 3.0**.

Increment Decrement

Lecture 6

Increment and Decrement Operators

Operator	Name	Description	Example (i = 1)	Read As
++var	Preincrement	Increment var by 1, and use the new var value in the statement	int j = ++ i; // j is 2, i is 2	i after increment, plus plus i, or i increased
var++	Postincrement	Increment var by 1, but use the original var value in the statement	int j = i ++; // j is 1, i is 2	i before increment, i plus plus, or i then increases
--var	Predecrement	decrement var by 1, and use the new var value in the statement	int j = -- i; // j is 0, i is 0	i after decrement, minus minus i , or i decreased
var--	Postdecrement	decrement var by 1, but use the original var value in the statement	int j = i --; // j is 1, i is 0	i before decrement, i minus minus or i then decreases

Increment and Decrement Operators, cont.

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

Increment and Decrement Operators, cont.

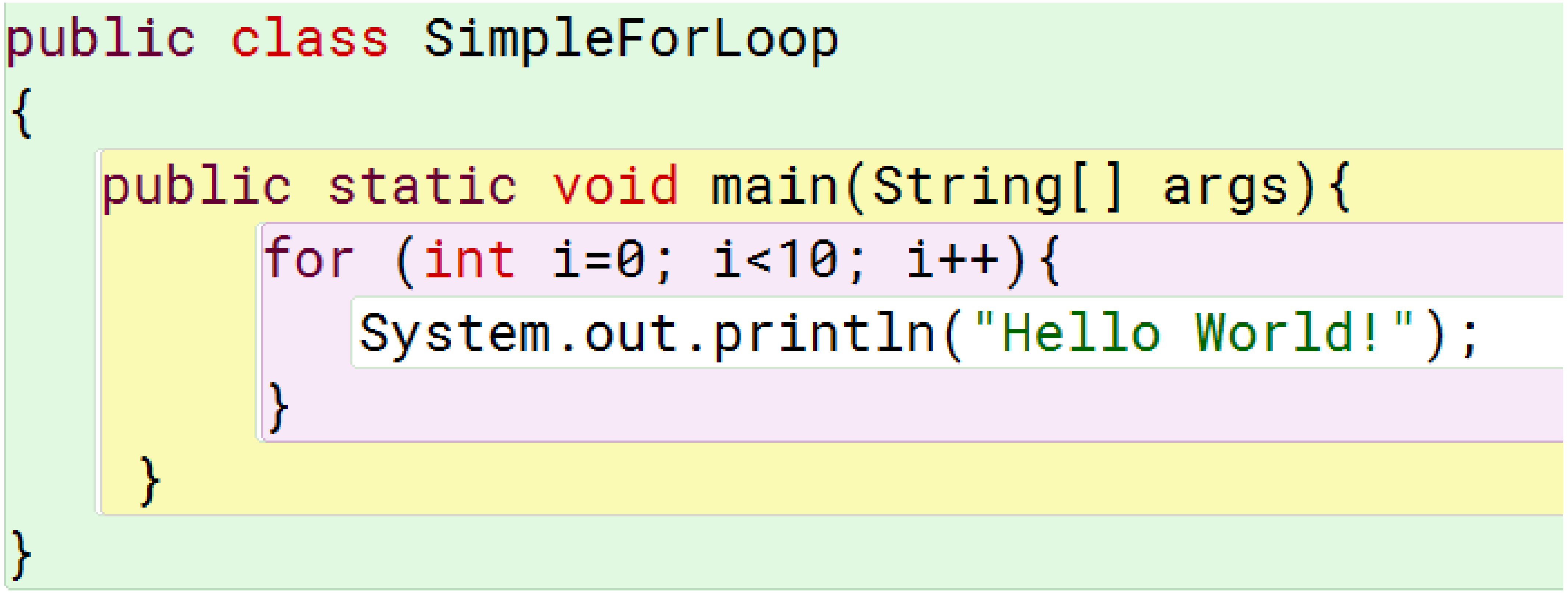
- Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read.
- Avoid using these operators in expressions that modify multiple variables, or the same variable for multiple times such as this:

```
int k = ++i + i.
```

Demonstration Program

SimpleforLoop.java

Using Increment for Counters



```
public class SimpleForLoop
{
    public static void main(String[] args){
        for (int i=0; i<10; i++){
            System.out.println("Hello World!");
        }
    }
}
```

Arithmetic Expression

Lecture 7

Arithmetic Expressions

$$\frac{3+4x}{5}-\frac{10(y-5)(a+b+c)}{x}+9(\frac{4}{x}+\frac{9+x}{y})$$

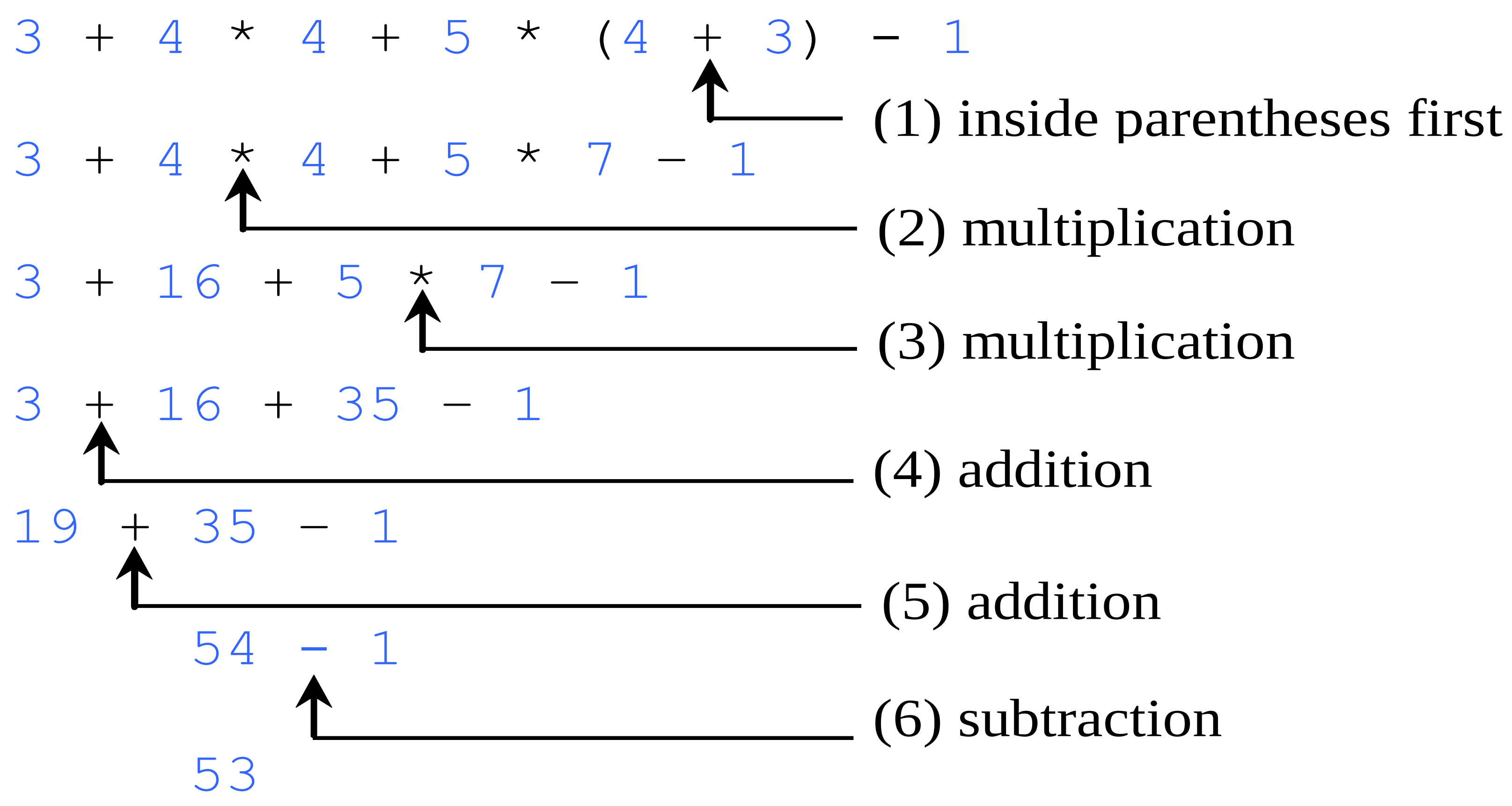
is translated to

```
(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)
```

How to Evaluate an Expression

- Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression are the same.
- Therefore, you can safely apply the arithmetic rule for evaluating a Java expression.

How to Evaluate an Expression



Operator Precedence

Lecture 8

Operator Precedence in Java

Java has well-defined rules for specifying the order in which the operators in an expression are evaluated when the expression has several operators. For example, multiplication and division have a higher precedence than addition and subtraction. Precedence rules can be overridden by explicit parentheses.

- **Precedence Order**
    - When two operators share an operand the operator with the higher *precedence* goes first. For example, 1 + 2 * 3 is treated as 1 + (2 * 3), whereas 1 * 2 + 3 is treated as (1 * 2) + 3 since multiplication has a higher precedence than addition.

Operator Precedence in Java

Associativity

- When an expression has two operators with the same precedence, the expression is evaluated according to its *associativity*.
- For example x = y = z = 17 is treated as x = (y = (z = 17)), leaving all three variables with the value 17, since the = operator has right-to-left associativity (and an assignment statement evaluates to the value on the right hand side).
- On the other hand, 72 / 2 / 3 is treated as(72 / 2) / 3 since the / operator has left-to-right associativity. Some operators are not associative: for example, the expressions (x <= y <= z) and x++-- are invalid.

Operator Precedence in Java

Precedence and associativity of Java operators

- The table below shows all Java operators from highest to lowest precedence, along with their associativity.
- Most programmers do not memorize them all, and even those that do still use parentheses for clarity.

Precedence

- **Parenthesis (bracket)**
- **Increment (Decrement)**
- **Sign (+/-) and negation (For logic !, ~)**
- **Multiplication/Division/Modulo**
- **Addition/Subtraction/String Concatenation**
- Bitwise Shift
- Inequality/Equality
- Bitwise Operator
- Bitwise and/or logical evaluation
- Conditional statement
- Assignments

Operator	Description	Level	Associativity
[]	access array element	1	left to right
.	access object member		
()	invoke a method		
++	post-increment		
--	post-decrement		
++	pre-increment	2	right to left
--	pre-decrement		
+	unary plus		
-	unary minus		
!	logical NOT		
~	bitwise NOT		
()	cast	3	right to left
new	object creation		
* / %	multiplicative	4	left to right
+ -	additive	5	left to right
+	string concatenation		

Operator	Description	Level	Associativity
`<<` `>>` `>>>`	shift	6	left to right
`<` `<=` `>` `>=` instanceof	relational type comparison	7	left to right
`==` `!=`	equality	8	left to right
`&`	bitwise AND	9	left to right
`^`	bitwise XOR	10	left to right
`|`	bitwise OR	11	left to right
`&&`	conditional AND	12	left to right
`||`	conditional OR	13	left to right
`?` `:`	conditional	14	right to left
`=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=` `>>>=`	assignment	15	right to left

Demonstration Program

SimpleforLoop.java

Augmented Assignments

Lecture 9

Assignments in Java

- An **assignment** in Java uses the assignment operator (=) to assign the result of an expression to a variable. In its simplest form, you code it like this:

```
variable = expression;
```

- For example:

```
int a = (b * c) / 4;
```

Side Effects (return value) of Assignments

- An assignment expression has a return value just as any other expression does; the return value is the value that’s assigned to the variable. For example, the return value of the expression a = 5 is 5. This allows you to create some interesting, but ill-advised, expressions by using assignment expressions in the middle of other expressions. For example:

  ```
  int a, b;
  a = (b = 3) * 2; // a is 6, b is 3
  ```
- Using assignment operators in the middle of an expression can make the expression harder to understand, so it’s not recommend that.

Augmented Assignments in Java

- A **augmented assignment** operator is an operator that performs a calculation and an assignment at the same time. All Java binary arithmetic operators (that is, the ones that work on two operands) have equivalent compound assignment operators:

Operator	Name	Example	Equivalent	Read As
+=	Addition Assignment	i += 8	i = i + 8	Increased by
-=	Subtraction Assignment	i -= 8	i = i – 8	Decreased by
*=	Multiplication Assignment	i *= 8	i = i * 8	Multiplied by
/=	Division Assignment	i /= 8	i = i / 8	Divided by
%=	Remainder Assignment	i %= 8	i = i % 8	Modded by

Augmented Assignment

- For example, the statement
```
a += 10;
```
- is equivalent to
```
a = a + 10;
```
- Technically, an assignment is an expression, not a statement. Thus, a = 5 is an assignment expression, not an assignment statement. It becomes an assignment statement only when you add a semicolon to the end.

Demonstration Program

AssignmentOperatorDemo.java

Data Type Conversion

Lecture 10

Numeric Type Conversions (Casting)

Floating point number can be converted into integers using explicit casting

- When an integer and a floating-point number are involved in a binary operation, Java automatically converts the integer to a floating-point value. So, 3 * 4.5 is same as 3.0 * 4.5. (This is automatic conversion.)

- You can always assign a value to a numeric value variable whose type supports a larger range of values: thus, for instance, you can assign a long value to a float variable.

Numeric Type Conversions (Casting)

Floating point number can be converted into integers using explicit casting

- **Casting** is an operation that converts a value of one data type into a value of another data type.

- Casting a type with a small range to a type with a larger range is known as **widening a type (upward casting)**.

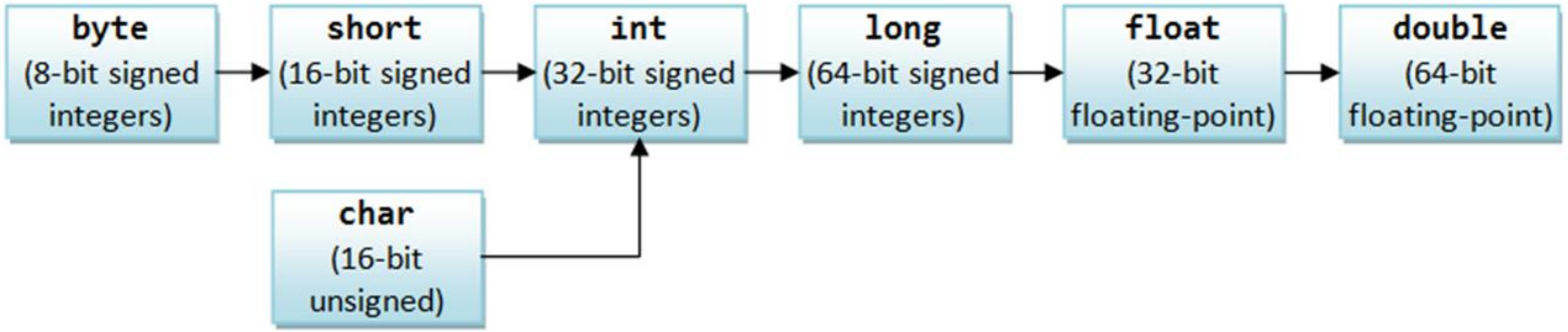
- Casting a type with a large range to a type with a smaller range is known as **narrowing a type (downward casting)**. Java will automatically widen a type, but you mist narrow a type explicitly.

Numeric Type Conversions (Casting)

Floating point number can be converted into integers using explicit casting

The syntax for casting a type is to specify the target type in parentheses, followed by the variable's name or the value to be case.

System.out.println((int) 1.7);



Orders of Implicit Type-Casting for Primitives

Numeric Type Conversion

Consider the following statements:

```
byte i = 100;

long k = i * 3 + 4;

double d = i * 3.1 + k / 2;
```

Conversion Rules (implicit casting)

- When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:
  1. If one of the operands is double, the other is converted into double.
  2. Otherwise, if one of the operands is float, the other is converted into float.
  3. Otherwise, if one of the operands is long, the other is converted into long.
  4. Otherwise, both operands are converted into int.

Type Casting

```
Implicit casting
  double d = 3; (type widening)

Explicit casting
  int i = (int)3.0; (type narrowing)
  int i = (int)3.9; (Fraction part is truncated)
```

What is wrong? int x = 5 / 2.0;

Problem: Keeping Two Digits After Decimal Points

- Write a program that displays the sales tax with two digits after the decimal point.
- Technique:

```
double num = 23.4567
(int) (num * 100) / 100.0;
```

(1) Step 1: (num * 100) shift two digit left. 2345.67
(2) Step 2: (int) casting and get rid of 67. 2345
(3) Step 3: shift right two digits. 23.45

Demonstration Program

SalesTax.java

Casting in an Augmented Expression

- In Java, an augmented expression of the form **x1 op= x2** is implemented as **x1 = (T)(x1 op x2)**, where **T** is the type for **x1**. Therefore, the following code is correct.

```
int sum = 0;
sum += 4.5; // sum becomes 4 after this statement
```

sum += 4.5 is equivalent to **sum = (int)(sum + 4.5)**.

Problem Solving and Example Programs

Lecture 11

Lab

Continued Fractional number

Lab: Continued Fractional Number

Conversion Fractional Number to Double

- Write a program class ContinuedFraction with a static main Method to show the decimal value of the continued fractional number below:
- (Hint: If you don’t know how to get started, you may download the ContinuedFraction.java from the download link:

$$4 + \frac{1}{2 + \frac{1}{6 + (1/7)}}$$

Project: ContinuedFraction.java

- Student should work on this project in Class.

Common Errors and Pitfalls in Numbers

Lecture 12

Common Errors and Pitfalls

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, untended integer division, and round-off errors.

- Common Error 1: Undeclared/Uninitialized Variables and Unused Variables.
- A variable must be declared with a type and assigned a value before using it. A Common error is not declaring or initializing a variable.

- `double interestRate = 0.05;`
- **`/* interestRate is different from interestrate */`**
- `double interest = interestrate * 45;`
- `/* Eclipse will catch this error */`

Common Errors and Pitfalls

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, untended integer division, and round-off errors.

- Common Error 2: Integer Overflow.
- byte a = 999;
- **/* byte can only store integer from -128 to 127. This will cause overflow. */**

Common Errors and Pitfalls

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, untended integer division, and round-off errors.

- Common Error 3: Round-off Errors.
- System.out.println(1.0 – 0.1 – 0.1 – 0.1 – 0.1 - 0.1);
- The system will display 0.5000000000000000001
- but not 0.5 because of round-off error or quantization error.

Common Errors and Pitfalls

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, untended integer division, and round-off errors.

- Common Error 4: Unintended Division.
- int number1 = 1;
- int number2 = 2;
- double average = (number1 + number2)/2;
- System.out.println(average);
- **/* this will cause unintended division error, (1+2)/2 = 1 */**

Common Pitfalls
Redundant Input Objects

- **System input = new Scanner(System.in);**
- System.out.print(“Enter an integer: ”);
- int v1 = input.nextInt();
- **System input1 = new Scanner(System.in);   /* redundant !!! Bad code */**
- System.out.print(“Enter an integer: ”);
- double v1 = input1.nextDouble();

Common Pitfalls

Redundant Input Objects

- You should use:
- **System input = new Scanner(System.in);**
- System.out.print(“Enter an integer: ”);
- int v1 = input.nextInt();

- System.out.print(“Enter an integer: ”);
- double f1 = input.nextDouble();

Lab

GetChange.java

Lab: GetChange.java

Calculate for a customer's change.

- When a customer comes to your store and purchase some product of some amount of money, which is defined as a double variable **cost**. He pays with another amount of money, which is defined as another double variable **amtPaid**.
- Write a program to calculate how many dollars, quarters, dimes, nickels and pennies he should receive as change. (The dollar amount does not have to specify what bills he will receive, while the coins should be specified.)
- (Hint: You do need to ask user's input for the two double variables in this program. Just specify
- cost=52.14;
- amtPaid = 100.0;

Project: GetChange.java

- Student should work on this project in Class.

Chapter Project: Display Current Time

Lecture 13

Chapter Project: DisplayCurrentTime.java

Chapter projects are unguided programming exercises.

- The purpose of having chapter project is to allow students to do programming on their own.
- I provide the background information and problem description.
- Only example program will be provided. I will not lead the program review step by step, because the program answer can have many formats and it is up to the programmer to perform at his own best based on his own domain knowledge to develop his programming style.

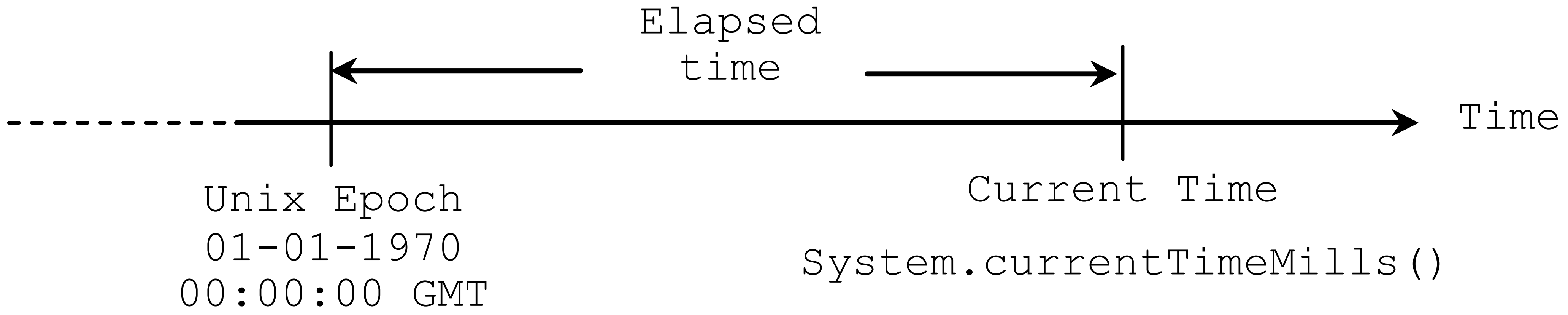
Problem: Displaying Current Time

- Write a program that displays current time in GMT in the format hour:minute:second such as 1:45:19, and your own local time if possible.
- I will use PST (Pacific Standard Time) as example.

Problem: Displaying Current Time

- The currentTimeMillis method in the System class returns the current time in milliseconds since the midnight, January 1, 1970 GMT. (1970 was the year when the Unix operating system was formally introduced.)
- You can use this method to obtain the current time, and then compute the current second, minute, and hour as follows.

Problem: Displaying Current Time



Java, UTC & System.currentTimeMillis()

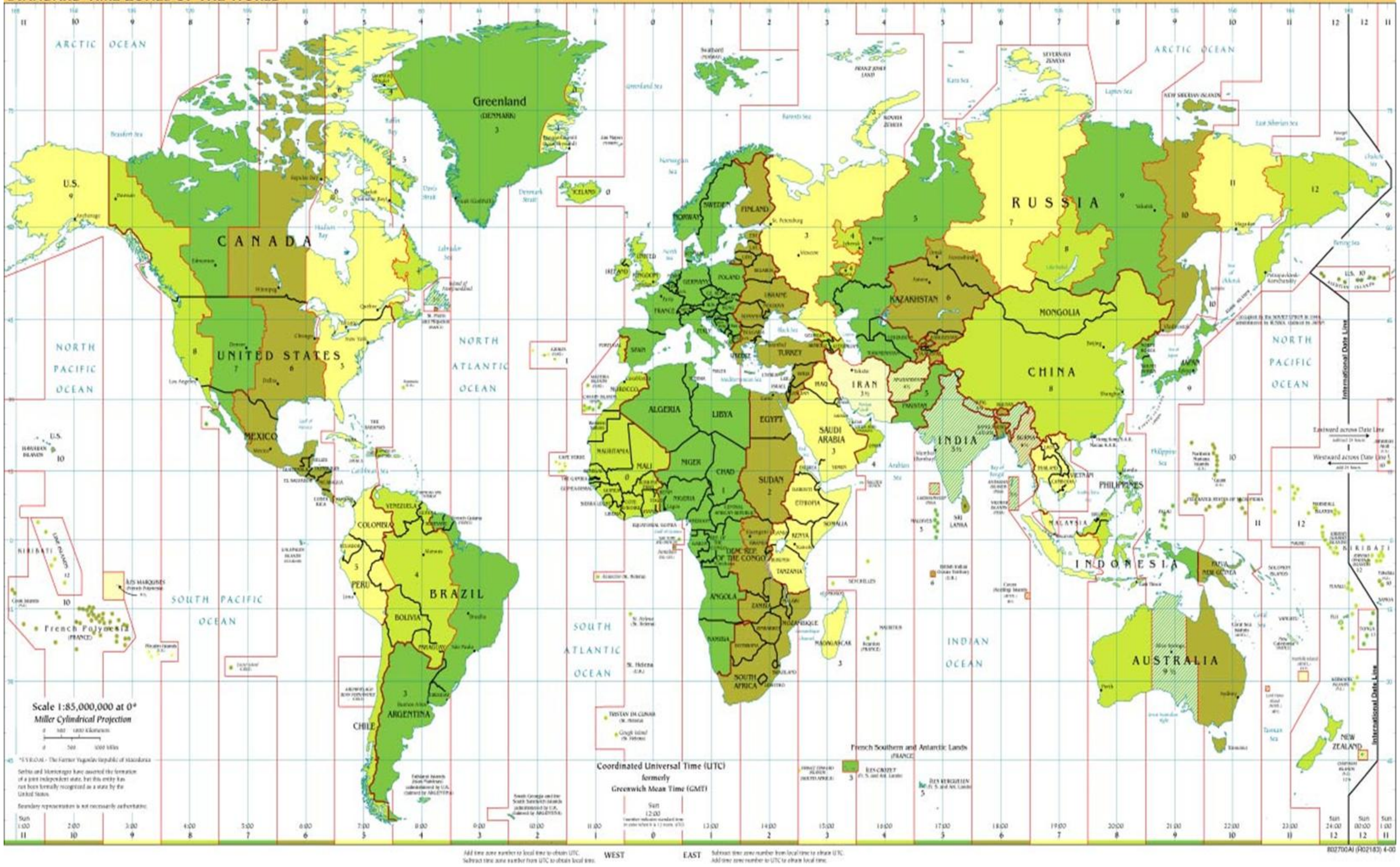
- **System.currentTimeMillis() example**

```
long millis = System.currentTimeMillis();

// prints a Unix timestamp in milliseconds
System.out.println(millis);

// prints the same Unix timestamp in seconds
System.out.println(millis / 1000);
```

Java, UTC & System.currentTimeMillis()

- **What exactly does currentTimeMillis() return?** System.currentTimeMillis() returns the number of milliseconds passed since a moment in time known as the Unix Epoch: **1970-01-01 00:00:00 UTC** (for most intents and purposes UTC can be used interchangeably with GMT, the zero longitude reference timezone). Note that this method is based on the time of the system/machine it's running on.

STANDARD TIME ZONES OF THE WORLD


ARCTIC OCEAN
Greenland
(DENMARK)
CANADA
UNITED STATES
U.S.
MEXICO
NORTH PACIFIC OCEAN
NORTH ATLANTIC OCEAN
SOUTH PACIFIC OCEAN
SOUTH ATLANTIC OCEAN
INDIAN OCEAN
ICELAND
FINLAND
SWEDEN
NORWAY
POLAND
UKRAINE
GERMANY
FRANCE
SPAIN
TURKEY
ALGERIA
LIBYA
EGYPT
NIGER
CHAD
SUDAN
ETHIOPIA
SOMALIA
KENYA
TANZANIA
DEM. REP. OF THE CONGO
ANGOLA
ZAMBIA
NAMIBIA
SOUTH AFRICA
MADAGASCAR
SAUDI ARABIA
IRAN
AFGHANISTAN
PAKISTAN
INDIA
SRI LANKA
KAZAKHSTAN
MONGOLIA
CHINA
RUSSIA
JAPAN
PHILIPPINES
MALAYSIA
INDONESIA
PAPUA NEW GUINEA
AUSTRALIA
NEW ZEALAND
KIRIBATI
French Polynesia (FRANCE)
COLOMBIA
VENEZUELA
PERU
BRAZIL
BOLIVIA
CHILE
ARGENTINA
International Date Line
Eastward across Date Line
Westward across Date Line
Scale 1:85,000,000 at 0°
Miller Cylindrical Projection
French Southern and Antarctic Lands (FRANCE)
Coordinated Universal Time (UTC) formerly Greenwich Mean Time (GMT)
Sun 12:00
WEST
EAST
Add time zone number to local time to obtain UTC. Subtract time zone number from UTC to obtain local time.
Subtract time zone number from local time to obtain UTC. Add time zone number to UTC to obtain local time.
802700AI (R02183) 4-00