# Think Java

CHAPTER 2: VARIABLES AND OPERATORS

DR. ERIC CHOU

IEEE SENIOR MEMBER

## Objectives

- This chapter describes how to write statements using variables, which store values like numbers and words, and operators, which are symbols that perform a computation. We also explain three kinds of programming errors and offer additional debugging advice.
- To run the examples in this chapter, you will need to create a new Java class with a main method (see Section 1.3). We often omit the class and method definitions to keep the examples concise.

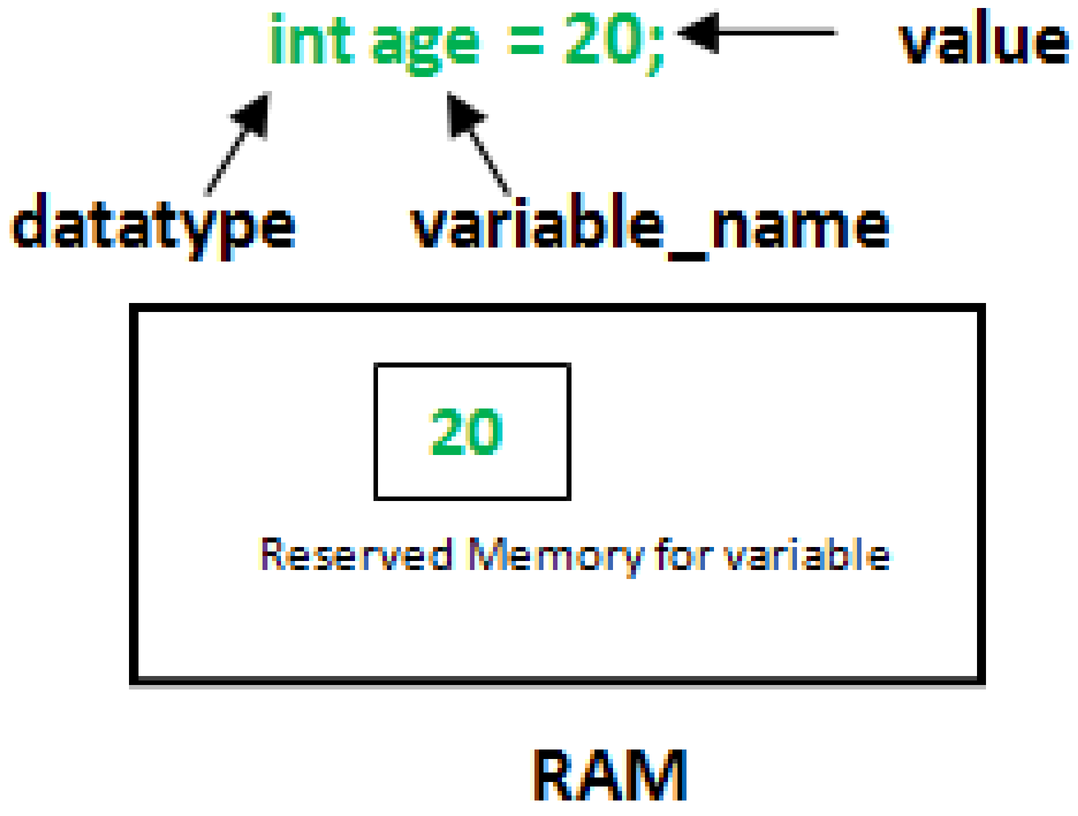
## Topics

- Variable and Memory
- Print out variables
- Operators and Expressions
- Compilation Errors

LECTURE 1

## Declaring variables

# Variable and Memory



## Variable Declaration

- To declare an integer variable named x, you simply type:

```
int x;
```

- Note that x is an arbitrary name for the variable. In general, you should use names that indicate what the variables mean.

```
String firstName;
String lastName;
int hour, minute;
```

## Variable Assignment

Now that we have declared some variables, we can use them to store values. We do that with an assignment statement.

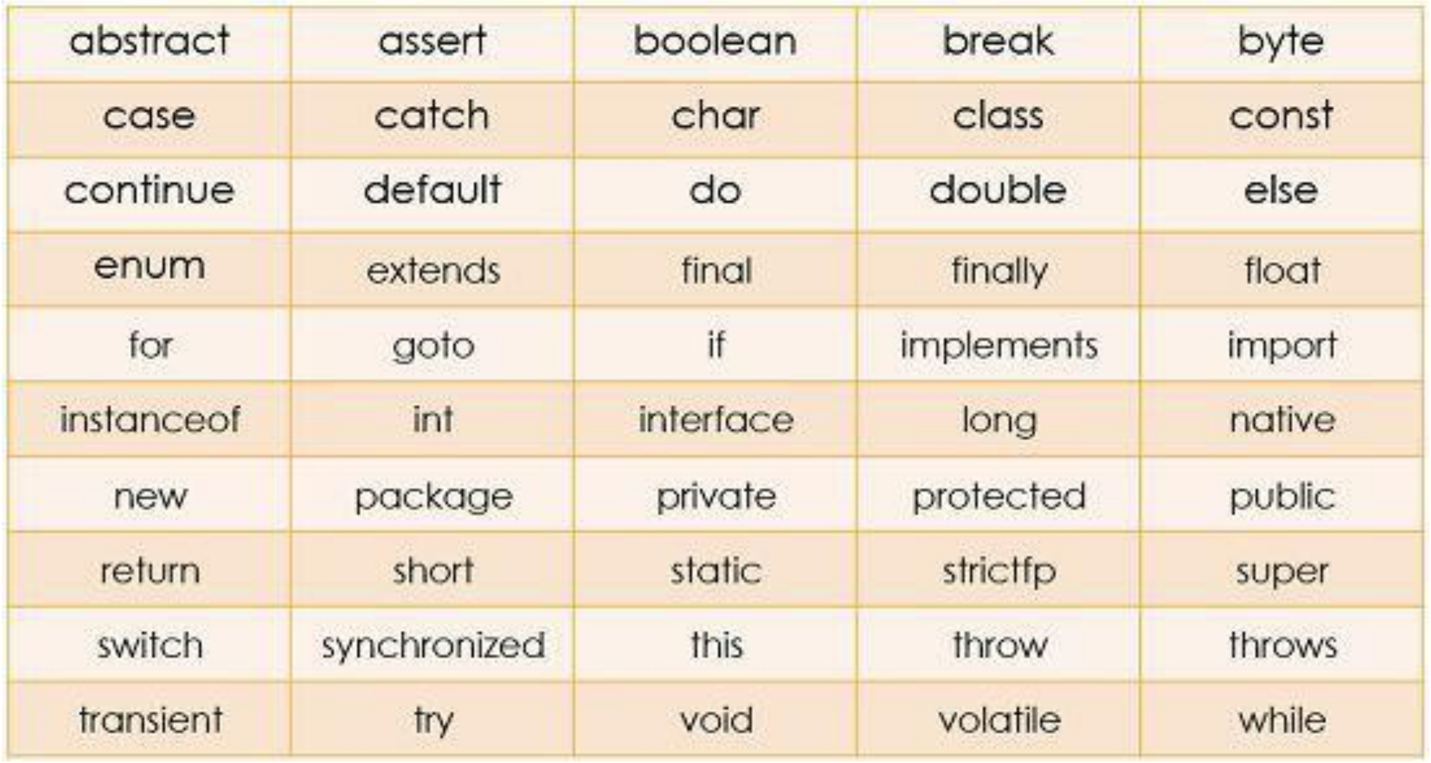
```
message = "Hello!";   // give message the value
"Hello!"
hour = 11;            // assign the value 11 to hour
minute = 59;          // set minute to 59
```

This example shows three assignments, and the comments illustrate different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a **named storage location**.
- When you make an assignment to a variable, you **update its value**.

## Java Keywords

- You can use any name you want for a variable. But there are about 50 reserved words, called keywords, that you are not allowed to use as variable names.
- These words include public, class, static, void, and int, which are used by the compiler to analyze the structure of the program.



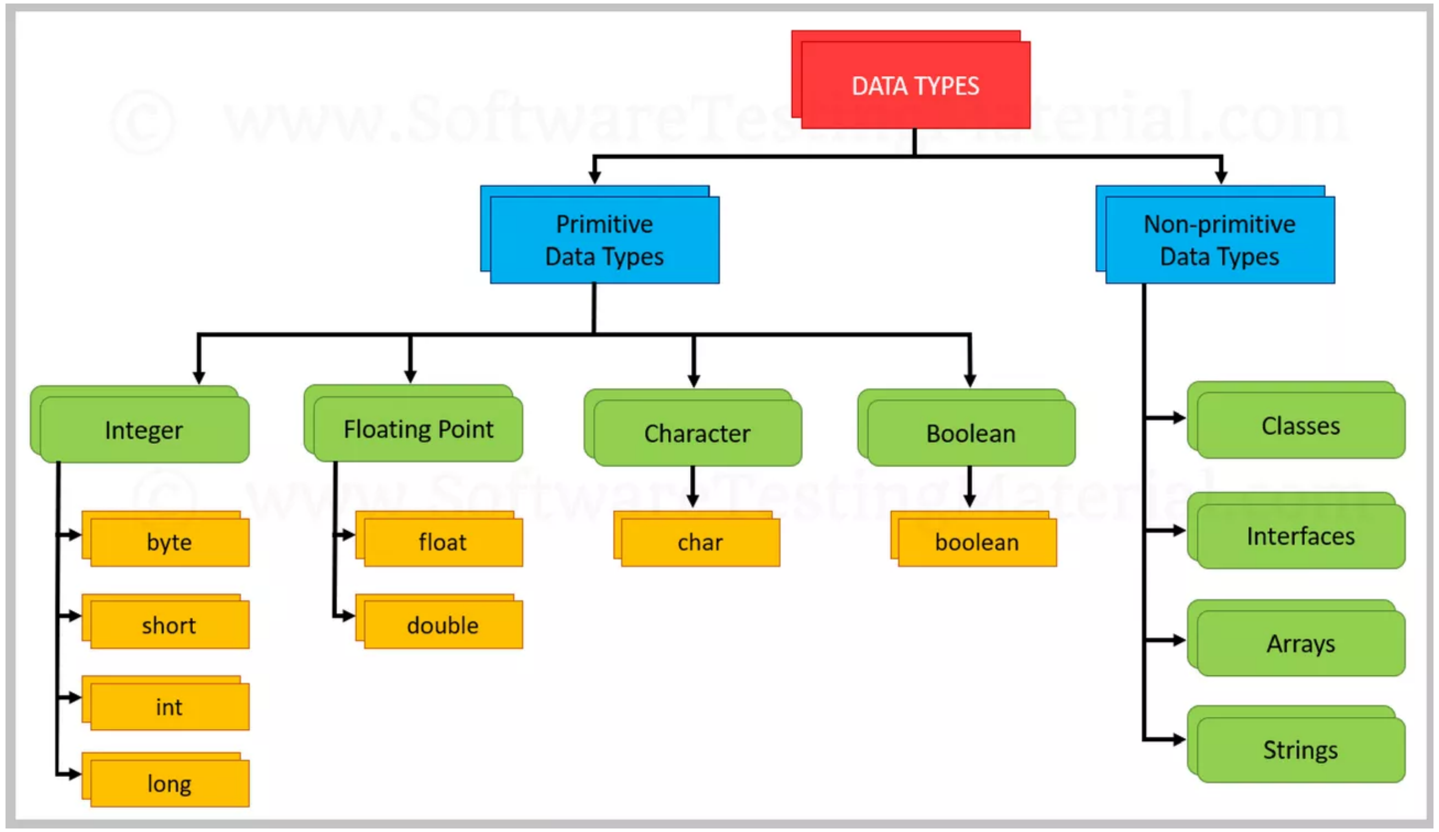
| | | | | |
|---|---|---|---|---|
| abstract | assert | boolean | break | byte |
| case | catch | char | class | const |
| continue | default | do | double | else |
| enum | extends | final | finally | float |
| for | goto | if | implements | import |
| instanceof | int | interface | long | native |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

LECTURE 2

## Data Types

## Data Types

int, double, char, boolean

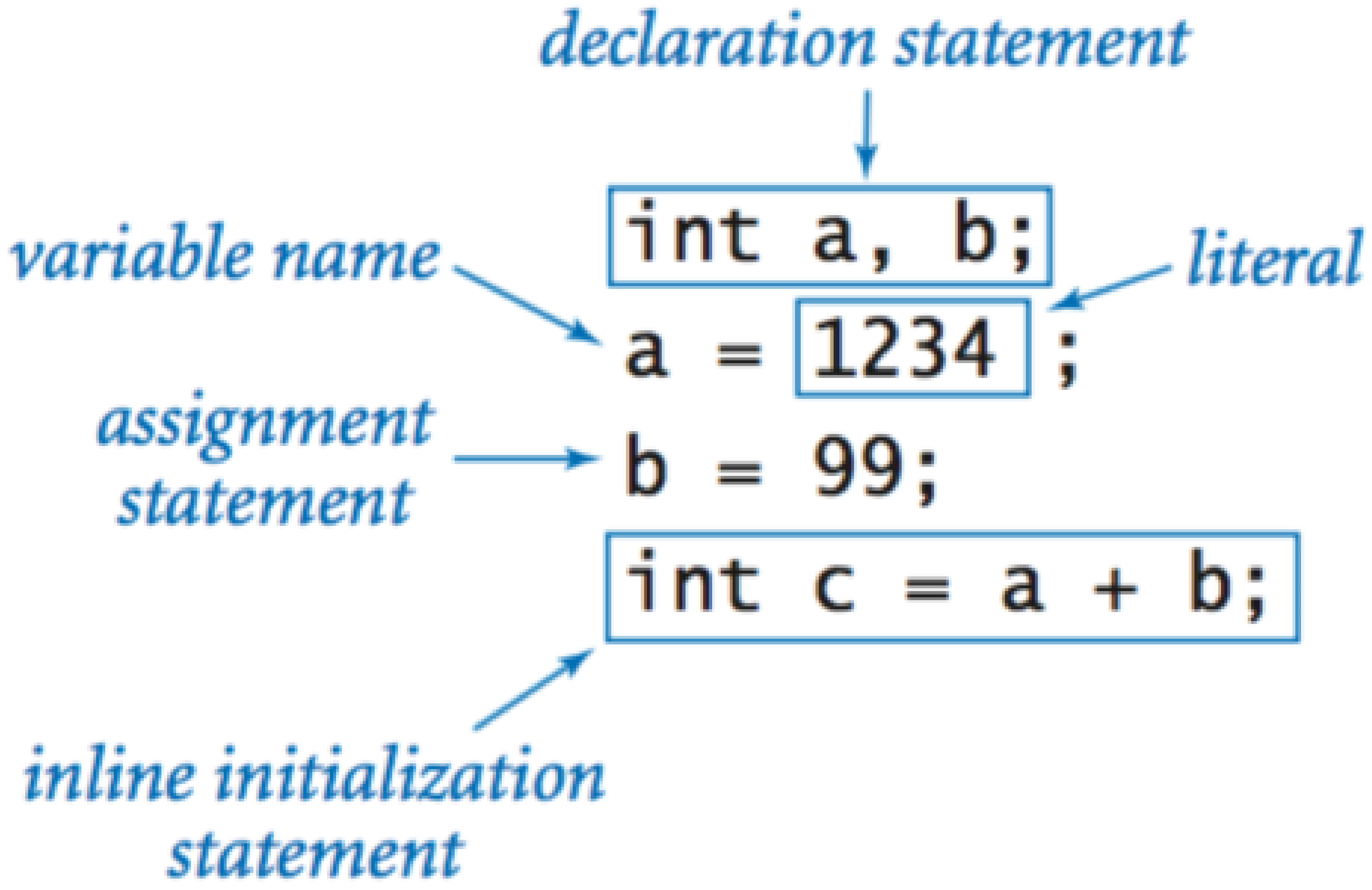


## PRIMITIVE DATA TYPE in JAVA

| Type | Contains | Default | Size | Range |
|---|---|---|---|---|
| byte | Signed integer | 0 | 8 bits | -128 to 127 |
| short | Signed integer | 0 | 16 bits | -32768 to 32767 |
| int | Signed integer | 0 | 32 bits | -2147483648 to 2147483647 |
| float | IEEE 754 floating point | 0.0f | 32 bits | ±1.4E-45 to ±3.4028235E+38 |
| long | Signed integer | 0L | 64 bits | -9223372036854775808 to 9223372036854775807 |
| double | IEEE 754 floating point | 0.0d | 64 bits | ±4.9E-324 to ±1.7976931348623157E+308 |
| | | | | |
| boolean | true or false | FALSE | 1 bit | NA |
| char | Unicode character | '\u0000' | 16 bits | \u0000 to \uFFFF |

LECTURE 3

## Memory diagrams



```
int a, b;
a = 1234 ;
b = 99;
int c = a + b;
```

## Variable Assignment

- In Java, an assignment statement can make two variables equal, but they don't have to stay that way.

```
int a = 5;
int b = a;       // a and b are now equal
a = 3;           // a and b are no longer equal
```

- The third line changes the value of a, but it does not change the value of b, so they are no longer equal.
- Taken together, the variables in a program and their current values make up the program's state. Figure 2.1 shows the state of the program after these assignment statements run.

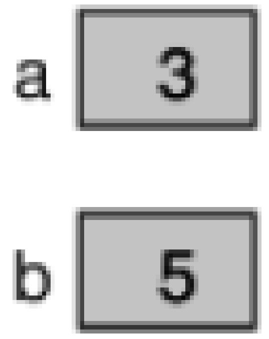


Figure 2.1: Memory diagram of the variables `a` and `b`.

LECTURE 4

## Printing variables

## Print out the value of a variable

- You can display the current value of a variable using print or println. The following statements declare a variable named firstLine, assign it the value "Hello, again!", and display that value.

```
String firstLine = "Hello, again!";
System.out.println(firstLine);
```

- When we talk about displaying a variable, we generally mean the value of the variable. To display the name of a variable, you have to put it in quotes.

```
System.out.print("The value of firstLine is ");
System.out.println(firstLine);
```

## Print out the value of a variable

- For this example, the output is:

  The value of **firstLine** is Hello, again!

- Conveniently, the code for displaying a variable is the same regardless of its type. For example:

```
int hour = 11;
int minute = 59;
System.out.print("The current time is ");
System.out.print(hour);
System.out.print(":");
System.out.print(minute);
System.out.println(".");
```

- The output of this program is:

```
The current time is 11:59.
```

LECTURE 5

## Arithmetic operators

## Operators

- **Operators** are symbols that represent simple computations. For example, the addition operator is +, subtraction is -, multiplication is *, and division is /.
- The following program converts a time of day to minutes:

```
int hour = 11;
int minute = 59;
System.out.print("Number of minutes since midnight: ");
System.out.println(hour * 60 + minute);
```

- The output is:

```
Number of minutes since midnight: 719
```

## Expression

- In this program, hour * 60 + minute is an **expression**, which represents a single value to be computed (719). When the program runs, each variable is replaced by its current value, and then the operators are applied. The values that operators work with are called **operands**.
- Expressions are generally a combination of numbers, variables, and operators. When compiled and executed, they become a single value. For example, the expression 1 + 1 has the value 2. In the expression hour - 1, Java replaces the variable with its value, yielding 11 - 1, which has the value 10.
- In the expression hour * 60 + minute, both variables get replaced, yielding 11 * 60 + 59. The multiplication happens first, yielding 660 + 59. Then the addition yields 719.

## Arithmetic Operators

- Addition, subtraction, and multiplication all do what you expect, but you might be surprised by division. For example, the following fragment tries to compute the fraction of an hour that has elapsed:

```
System.out.print("Fraction of the hour that has
passed: ");
System.out.println(minute / 60);
```

- The output is:

```
Fraction of the hour that has passed: 0
```

- This result often confuses people. The value of minute is 59, and 59 divided by 60 should be 0.98333, not 0. The problem is that Java performs “integer division” when the operands are integers. By design, integer division always rounds toward zero, even in cases like this one where the next integer is close.

## Rounding

- As an alternative, we can calculate a percentage rather than a fraction:

```
System.out.print("Percent of the hour that
has passed: ");
System.out.println(minute * 100 / 60);
```

- The new output is:

```
Percent of the hour that has passed: 98
```

- Again the result is rounded down, but at least now it’s approximately correct.

LECTURE 6

## Floating-point numbers

# Floating Point Operation

- A more general solution is to use floating-point numbers, which can represent fractions as well as integers. In Java, the default floating-point type is called double, which is short for double-precision. You can create double variables and assign values to them the same way we did for the other types:

```
double pi;
pi = 3.14159;
```

- Java performs "floating-point division" when one or more operands are double values. So we can solve the problem we saw in the previous section:

```
double minute = 59.0;
System.out.print("Fraction of the hour that has passed: ");
System.out.println(minute / 60.0);
```

- The output is:

```
Fraction of the hour that has passed: 0.98333333333333333
```

# Floating Point Operation

- Although floating-point numbers are useful, they can be a source of confusion. For example, Java distinguishes the integer value 1 from the floating-point value 1.0, even though they seem to be the same number. They belong to different data types, and strictly speaking, you are not allowed to make assignments between types.
- The following is illegal because the variable on the left is an int and the value on the right is a double:

```
int x = 1.1;  // compiler error
```

- It is easy to forget this rule, because in many cases Java automatically converts from one type to another:

```
double y = 1;  // legal, but bad style
```

- The preceding example should be illegal, but Java allows it by converting the int value 1 to the double value 1.0 automatically. This leniency is convenient, but it often causes problems for beginners. For example:

```
double y = 1 / 3;  // common mistake
```

## Floating Point Operation

- You might expect the variable y to get the value 0.333333, which is a legal floating-point value. But instead it gets the value 0.0. The expression on the right divides two integers, so Java does integer division, which yields the int value 0. Converted to double, the value assigned to y is 0.0.
- One way to solve this problem (once you figure out the bug) is to make the right-hand side a floating-point expression. The following sets y to 0.333333, as expected:

```
    double y = 1.0 / 3.0;   // correct
```

- As a matter of style, you should always assign floating-point values to floating-point variables. The compiler won't make you do it, but you never know when a simple mistake will come back and haunt you.

LECTURE 7

## Rounding errors

## Rounding Errors of Floating Points

- Most floating-point numbers are only approximately correct. Some numbers, like reasonably-sized integers, can be represented exactly. But repeating fractions, like 1/3, and irrational numbers, like π, cannot. To represent these numbers, computers have to round off to the nearest floating-point number.
- The difference between the number we want and the floating-point number we get is called rounding error. For example, the following two statements should be equivalent:

```
System.out.println(0.1 * 10);
System.out.println(0.1 + 0.1 + 0.1 + 0.1 + 0.1
                 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1);
```

- But on many machines, the output is:

```
1.0
0.9999999999999999
```

# Rounding Errors of Floating Points

The problem is that 0.1, which is a terminating fraction in decimal, is a repeating fraction in binary. So its floating-point representation is only approximate. When we add up the approximations, the rounding errors accumulate.

For many applications, like computer graphics, encryption, statistical analysis, and multimedia rendering, floating-point arithmetic has benefits that outweigh the costs. But if you need absolute precision, use integers instead. For example, consider a bank account with a balance of $123.45:

```
double balance = 123.45;  // potential rounding error
```

In this example, balances will become inaccurate over time as the variable is used in arithmetic operations like deposits and withdrawals. The result would be angry customers and potential lawsuits. You can avoid the problem by representing the balance as an integer:

```
int balance = 12345;      // total number of cents
```

This solution works as long as the number of cents doesn't exceed the largest integer, which is about 2 billion.

LECTURE 8

## Operators for strings

## String Concatenation

- In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following expressions are illegal:
  - `"Hello" - 1      "World" / 123      "Hello" * "World"`
- The + operator works with strings, but it might not do what you expect. For strings, the + operator performs concatenation, which means joining end-to-end. So "Hello, " + "World!" yields the string "Hello, World!".
- Likewise if you have a variable called name that has type String, the expression "Hello, " + name appends the value of name to the hello string, which creates a personalized greeting.

## String Concatenation

- Since addition is defined for both numbers and strings, Java performs automatic conversions you may not expect:

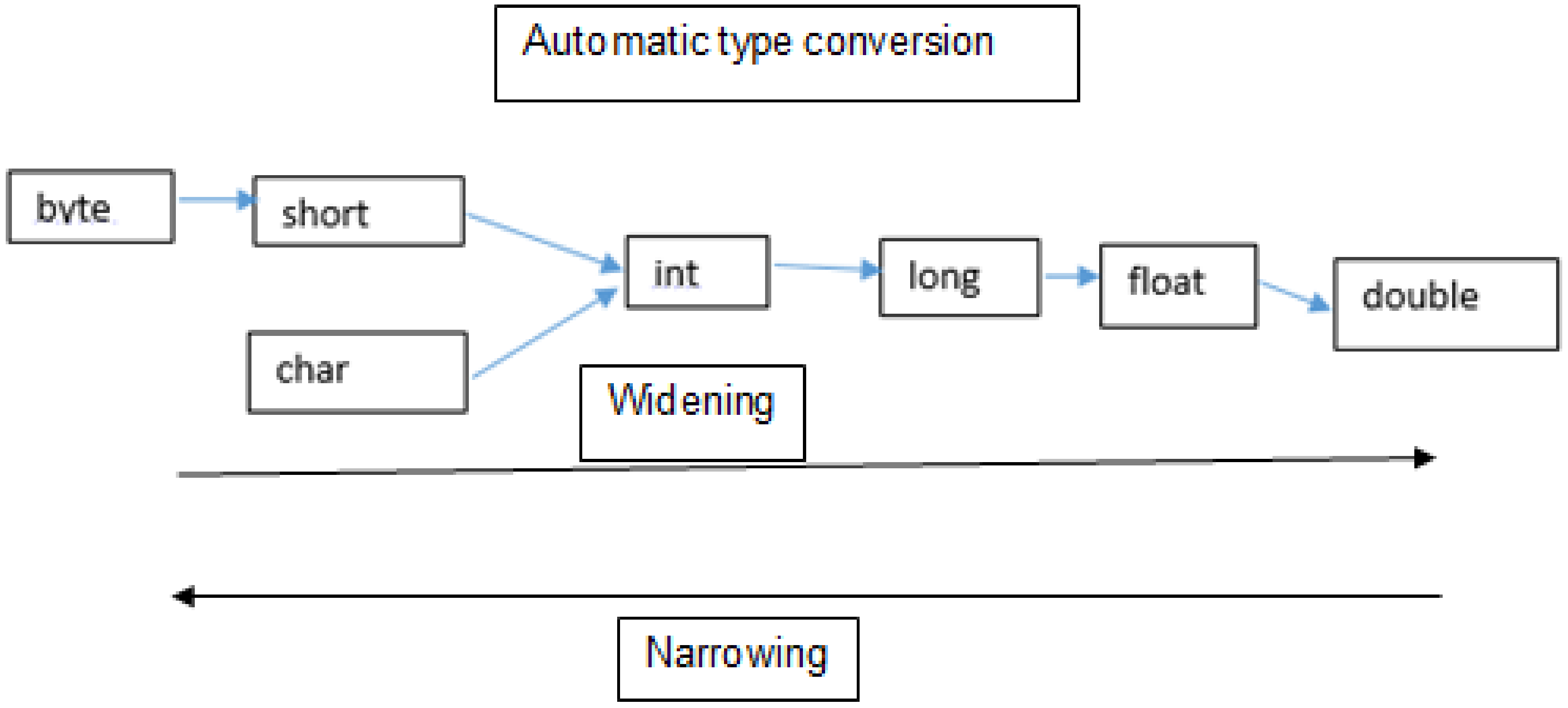
```
System.out.println(1 + 2 + "Hello");
// the output is 3Hello

System.out.println("Hello" + 1 + 2);
// the output is Hello12
```

- Java executes these operations from left to right. In the first line, 1 + 2 is 3, and 3 + "Hello" is "3Hello". But in the second line, "Hello" + 1 is "Hello1", and "Hello1" + 2 is "Hello12".

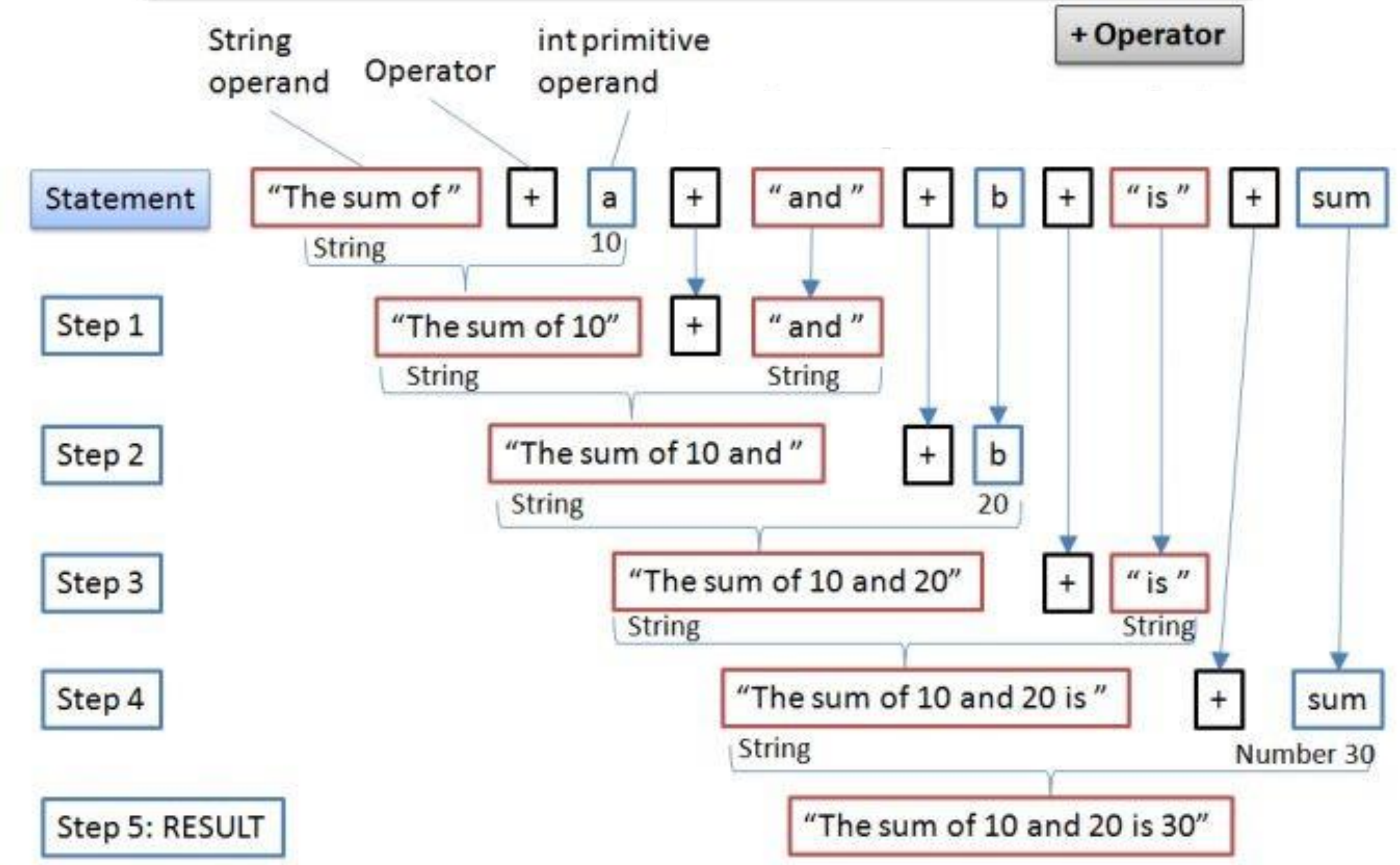
## String Concatenation

When more than one operator appears in an expression, they are evaluated according to the order of operations. Generally speaking, Java evaluates operators from left to right (as we saw in the previous section). But for numeric operators, Java follows mathematical conventions:

- Multiplication and division take "precedence" over addition and subtraction, which means they happen first. So 1 + 2 * 3 yields 7, not 9, and 2 + 4 / 2 yields 4, not 3.
- If the operators have the same precedence, they are evaluated from left to right. So in the expression minute * 100 / 60, the multiplication happens first; if the value of minute is 59, we get 5900 / 60, which yields 98. If these same operations had gone from right to left, the result would have been 59 * 1, which is incorrect.
- Any time you want to override the order of operations (or you are not sure what it is) you can use parentheses. Expressions in parentheses are evaluated first, so (1 + 2) * 3 is 9. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even though it doesn't change the result.


Automatic type conversion
byte
short
char
int
long
float
double
Widening
Narrowing

# String Concatenation with Primitive Data Types



If either operand of + is a String, then the + operator becomes a String Concatenation

## Demonstration of Data Operations

LECTURE 9

## Compiler error messages

## Compiler error messages

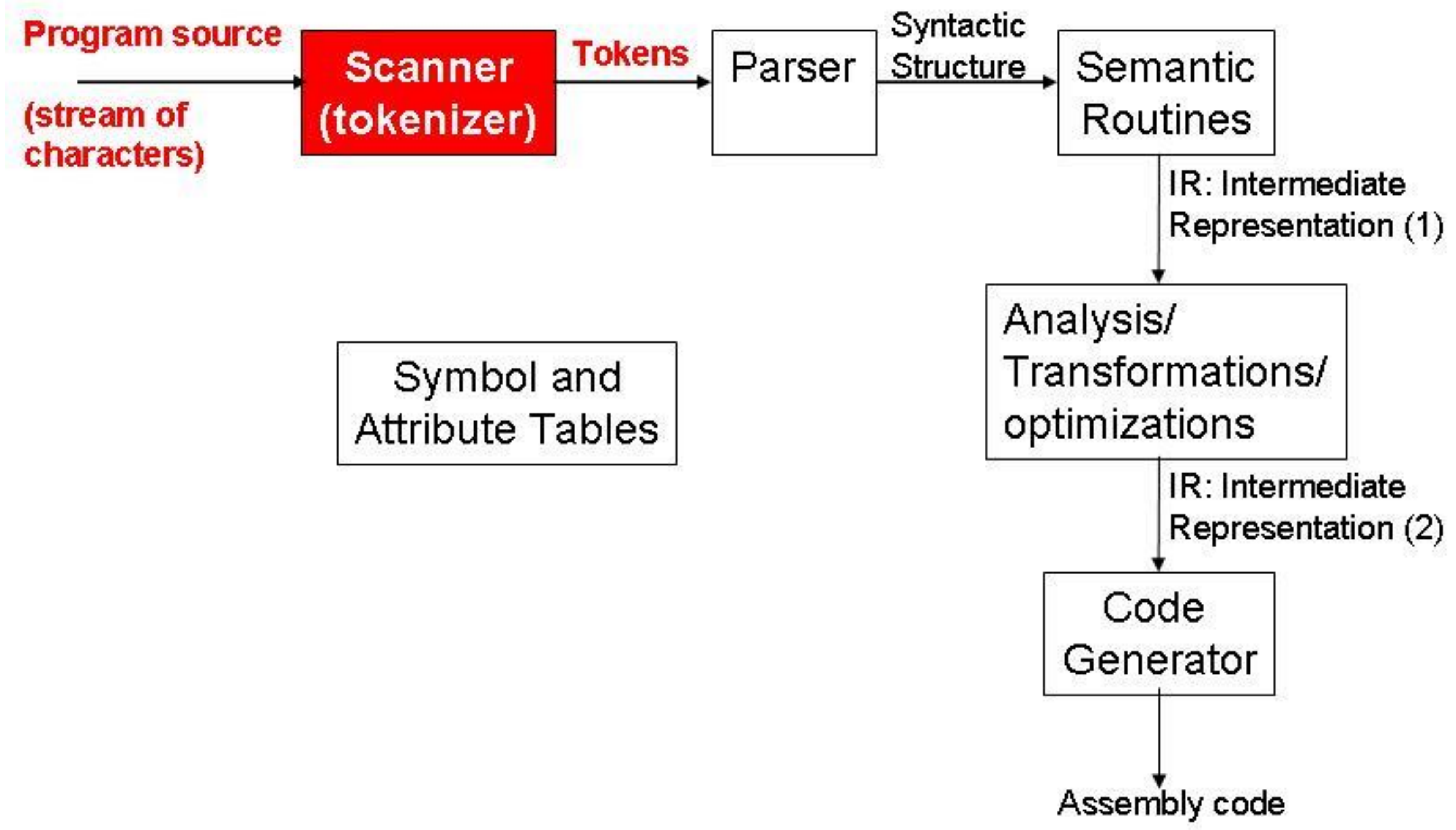
- Three kinds of errors can occur in a program: compile-time errors, run-time errors, and logic errors. It is useful to distinguish among them in order to track them down more quickly.
- Compile-time errors occur when you violate the rules of the Java language. For example, parentheses and braces have to come in matching pairs. So (1 + 2) is legal, but 8) is not. In the latter case, the program cannot be compiled, and the compiler displays an error.

# Lab

COMPILER ERRORS

## Compiler error messages

- There are two problems here. First, the error message is written from the compiler's point of view, not yours. Parsing is the process of reading a program before translating; if the compiler gets to the end of the file while still parsing, that means something was omitted. But the compiler doesn't know what. It also doesn't know where. The compiler discovers the error at the end of the program (line 7), but the missing brace should be on the previous line.
- Error messages contain useful information, so you should make an effort to read and understand them. But don't take them too literally. During the first few weeks of your programming career, you will probably spend a lot of time tracking down compile-time errors. As you gain experience, you will make fewer mistakes and find them more quickly

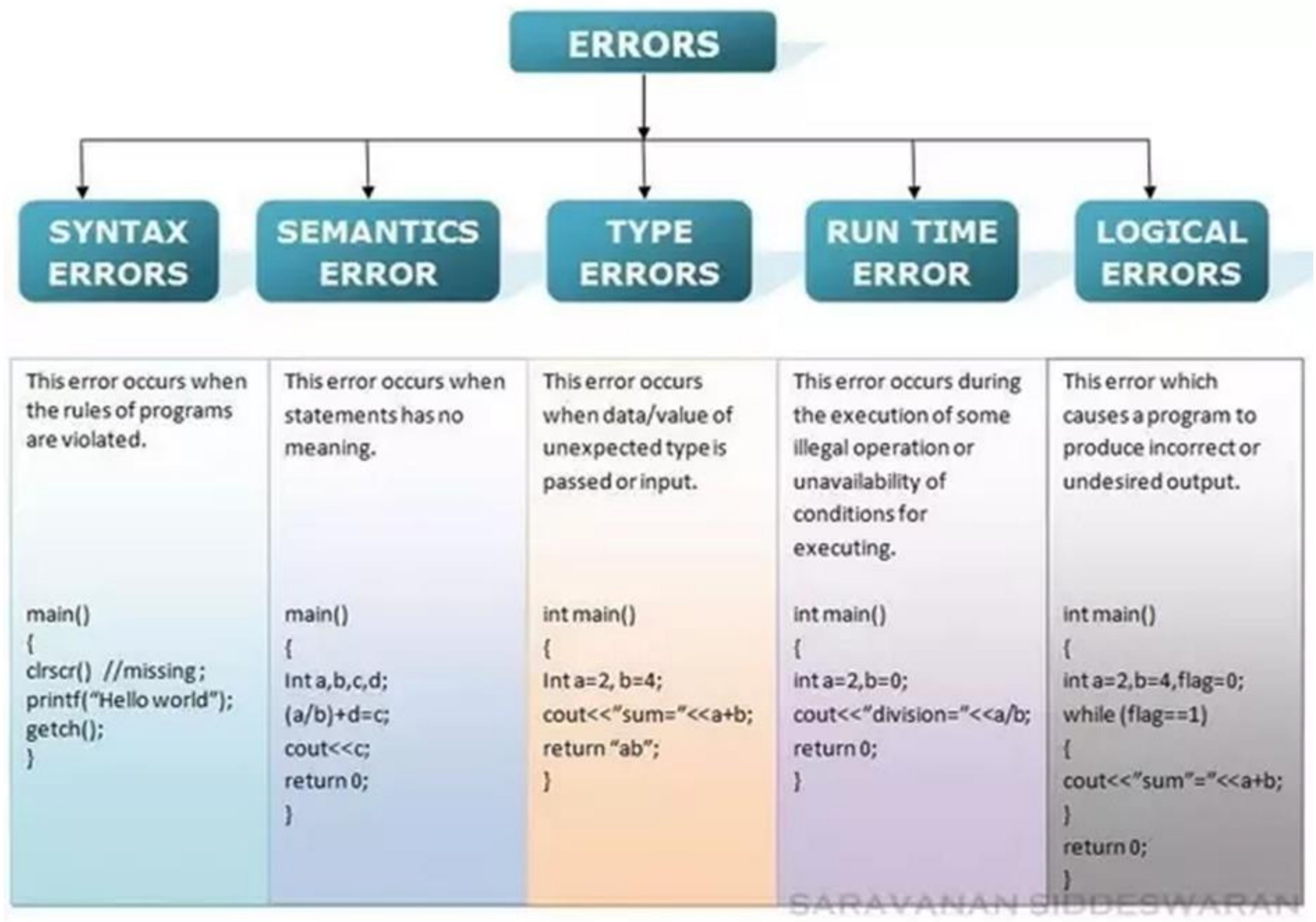

Program source
(stream of characters)
Scanner (tokenizer)
Tokens
Parser
Syntactic Structure
Semantic Routines
IR: Intermediate Representation (1)
Analysis/ Transformations/ optimizations
IR: Intermediate Representation (2)
Code Generator
Assembly code
Symbol and Attribute Tables

LECTURE 10

## Other types of errors

## Program Errors

- Compile-Time Errors (Syntax Error, Semantics Error, and Type Error)
  - Found during compilation
  - Usually caused by incorrect capitalization or spelling mistakes
- Run-Time Errors
  - Found during execution
  - Usually caused by invalid data such as dividing by zero
- Logic Errors
  - Found during testing
  - Usually caused by human error such as using the wrong equation, wrong strategy, etc.



## ERRORS

### SYNTAX ERRORS

This error occurs when the rules of programs are violated.

```
main()
{
clrscr()  //missing ;
printf("Hello world");
getch();
}
```

### SEMANTICS ERROR

This error occurs when statements has no meaning.

```
main()
{
Int a,b,c,d;
(a/b)+d=c;
cout<<c;
return 0;
}
```

### TYPE ERRORS

This error occurs when data/value of unexpected type is passed or input.

```
int main()
{
Int a=2, b=4;
cout<<"sum="<<a+b;
return "ab";
}
```

### RUN TIME ERROR

This error occurs during the execution of some illegal operation or unavailability of conditions for executing.

```
int main()
{
int a=2,b=0;
cout<<"division="<<a/b;
return 0;
}
```

### LOGICAL ERRORS

This error which causes a program to produce incorrect or undesired output.

```
int main()
{
int a=2,b=4,flag=0;
while (flag==1)
{
cout<<"sum"="<<a+b;
}
return 0;
}
```

## In-Class Demonstration Program

- Syntax Error
- Run-Time Error
- Logic Error

HELLO.JAVA

## Homework

## Homework

- Textbook Exercise: 2.1, 2.2, 2-3
- Project 2