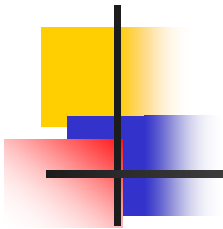
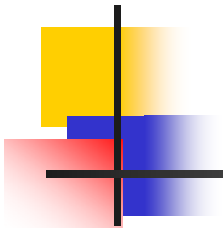
Expressions and Assignment



Outline

- Declaration Statements
- Primitive Data Types
- Identifiers
- Assignment Statements
- Arithmetic Operators and Expressions
- Precedence Rules
- Round-Off Errors in Floating-Point Numbers
- Integer and Floating-Point Division
- The % Operator
- Type Casting
- Increment and Decrement Operators



- Declaration Statements

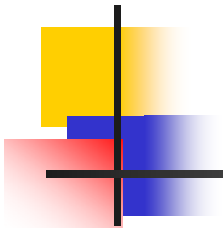
- Every variable in a Java program must be *declared* before it is used
  - A variable declaration tells the compiler what kind of data (type) will be stored in the variable
  - The type of the variable is followed by one or more variable names separated by commas, and terminated with a semicolon
  - 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*

```
int numberOfBeans;
double oneWeight, totalWeight;
```

- Primitive Data Types

Display 1.2 Primitive Types

TYPE NAME	KIND OF VALUE	MEMORY USED	SIZE RANGE
`boolean`	`true` or `false`	1 byte	not applicable
`char`	single character (Unicode)	2 bytes	all Unicode characters
`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
`float`	floating-point number	4 bytes	$-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$
`double`	floating-point number	8 bytes	$\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$

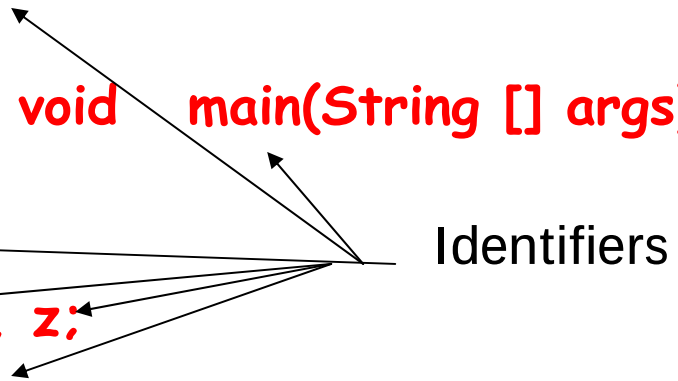


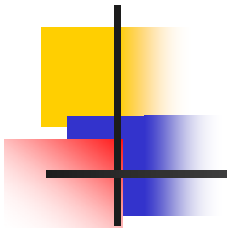
- Identifiers ...

- **`e.g. ➔ int variable = 3;`**
- Java statement can contain one or more identifiers.
- *Identifier*: 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 the underscore symbol
  - Java identifiers can theoretically be of any length
  - Java is a case-sensitive language: **`Rate`**, **`rate`**, and **`RATE`** are the names of three different variables

... - Identifiers ...

```
public class Example
{
    public static void main(String [] args)
    {
        int x;
        double y, z;
        String s;
        x = 4;
        y = 6.5;
        z = x + y;
        s = "ICS"
        System.out.println(s + w);
    }
}
```





... - Identifiers ...

- Keywords and Reserved words: Identifiers that have a predefined meaning in Java
  - Do not use them to name anything else

public **class** **void** **static**

- Predefined identifiers: Identifiers that are defined in libraries required by the Java language standard
  - Although they can be redefined, this could be confusing and dangerous if doing so would change their standard meaning

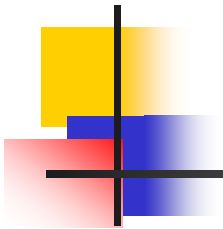
System **String** **println**

… - Identifiers

```
public class Example
{
    public static void   main(String [] args)
    {
        int x;
        double y, z;
        String s;
        x = 4;
        y = 6.5;
        z = x + y;
        s = "ICS"
        System.out.println(s + w);
    }
}
```

Keywords (Reserved identifiers)

Predefined identifiers



-- Naming Conventions (Leave it to the Lab)

- Start the names of variables, classes, methods, and objects with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters

  topSpeed bankRate1 timeOfArrival

- Start the names of classes with an uppercase letter and, otherwise, adhere to the rules above

  FirstProgram MyClass String

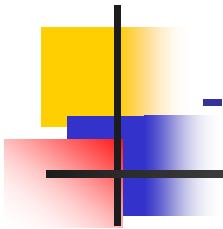
- Assignment Statements

- In Java, the assignment statement is used to change the value of a variable
  - The equal sign (=) is used as the assignment operator
  - An assignment statement consists of a variable on the left side of the operator, and an *expression* on the right side of the operator

```
Variable = Expression;
```

  - An *expression* consists of a variable, number, or mix of variables, numbers, operators, and/or method invocations

```
temperature = 98.6;
count = numberOfBeans;
```



-- Assignment Statements With Primitive Types

- When an assignment statement is executed, the expression is first evaluated, and then the variable on the left-hand side of the equal sign is set equal to the value of the expression

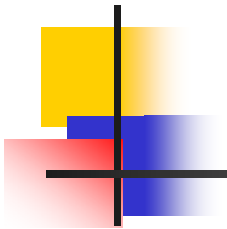
```
distance = rate * time;
```

- Note that a variable can occur on both sides of the assignment operator

```
count = count + 2;
```

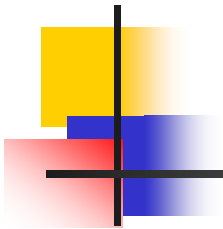
- The assignment operator is automatically executed from right-to-left, so assignment statements can be chained

```
number2 = number1 = 3;
```



-- Initialize Variables ...

- A variable that has been declared but that has not yet been given a value by some means is said to be *uninitialized*
- In certain cases an uninitialized variable is given a default value
  - It is best not to rely on this
  - Explicitly initialized variables have the added benefit of improving program clarity



... -- Initialize Variables

- The declaration of a variable can be combined with its initialization via an assignment statement

```
int count = 0;
double distance = 55 * .5;
char grade = 'A';
```

  - Note that some variables can be initialized and others can remain uninitialized in the same declaration

```
int initialCount = 50, finalCount;
```

-- Shorthand Assignment Statements ...

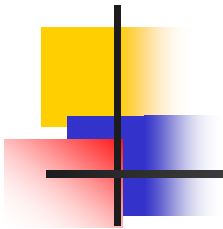
- **`e.g. ➔ variable += 3;`**
- Shorthand assignment notation combines the *assignment operator* (**=**) and an *arithmetic operator*
- It is used to change the value of a variable by adding, subtracting, multiplying, or dividing by a specified value
- The general form is

Variable Op = Expression

which is equivalent to

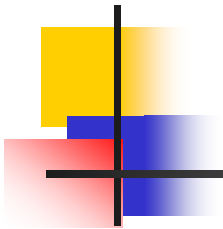
Variable = Variable Op (Expression)

  - The **`Expression`** can be another variable, a constant, or a more complicated expression
  - Some examples of what **`Op`** can be are **`+`**, **`-`**, **`*`**, **`/`**, or **`%`**



... -- 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;`
`amount *= count1 + count2;`	`amount = amount * (count1 + count2);`



-- Assignment Compatibility ...

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

```
int x = 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 variable = 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

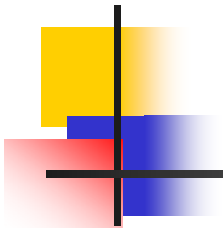
  `byte`→**`short`**→**`int`**→**`long`**→**`float`**→**`double`**

  `char` 

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

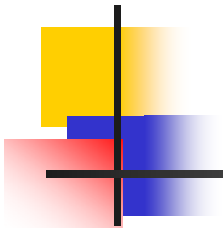
- An explicit *type cast* is required to assign a value of one type to a variable whose type appears to the left of it on the above list (e.g., **`double`** to **`int`**)

- 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`**



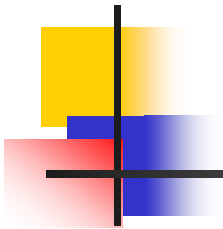
- Constants ... (Can be described with types)

- *Constant* (or *literal*): An item in Java which has one specific value that cannot change
  - Constants of an integer type may not be written with a decimal point (e.g., `10`)
  - Constants of a floating-point type can be written in ordinary decimal fraction form (e.g., `367000.0` or `0.000589`)
  - Constant of a floating-point type can also be written in *scientific* (or *floating-point*) *notation* (e.g., `3.67e5` or `5.89e-4`)
    - Note that the number before the `e` may contain a decimal point, but the number after the `e` may not



... - Constants

- Constants of type **`char`** are expressed by placing a single character in single quotes (e.g., **`'Z'`**)
- Constants for strings of characters are enclosed by double quotes (e.g., **`"Welcome to Java"`**)
- There are only two **`boolean`** type constants, **`true`** and **`false`**
  - Note that they must be spelled with all lowercase letters



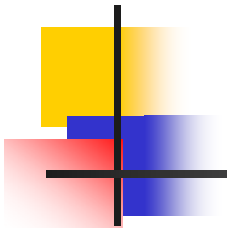
- Arithmetic Operators and Expressions ...

- As in most languages, Java *expressions* can be formed using variables, constants, and arithmetic operators
  - These operators are **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division), and **%** (modulo, remainder)
  - An expression can be used anyplace it is legal to use a value of the type produced by the expression

... - 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`**



- Precedence Rules ...

- An expression can be *fully parenthesized* in order to specify exactly what subexpressions 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
  - However, it's best (and sometimes necessary) to include them

... - Precedence Rules ...

High precedence

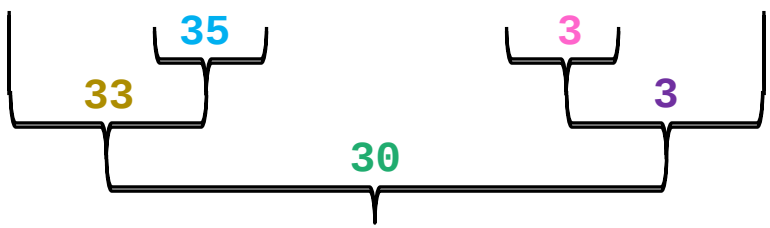
First: The uniray operators: +, -, ++, --, and !

Second: The binary arithmetic operators: *, /. And %

Third: The binary arithmatic operators: + and -

Low precedence

```
e.g. ➔ int x = y = -2 + 5 * 7 – 3 / 2 % 5;
This will be evaluated as..
   int x = (y = ((-2 + (5 * 7)) – ((7 / 2) % 5)));
```

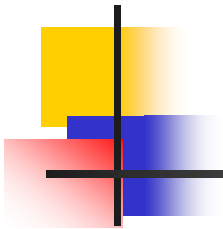


... - Precedence Rules ... (example is enough)

- When the order of two adjacent operations must be determined, the operation of higher precedence (and its apparent arguments) is grouped before the operation of lower precedence

  `base + rate * hours` is evaluated as
  `base + (rate * hours)`

- When two operations have equal precedence, the order of operations is determined by *associativity* rules



... - Precedence Rules

- Unary operators of equal precedence are grouped right-to-left

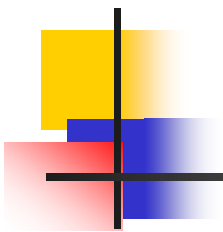
  `+-+rate` is evaluated as `+(-(+rate))`

- Binary operators of equal precedence are grouped left-to-right

  `base + rate + hours` is evaluated as
  `(base + rate) + hours`

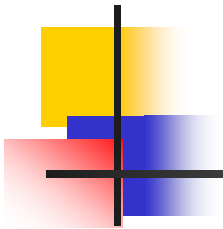
- Exception: A string of assignment operators is grouped right-to-left

  `n1 = n2 = n3;` is evaluated as `n1 = (n2 = n3);`



- Round-Off Errors in Floating-Point Numbers (NO Need for this!)

- Floating point numbers are only approximate quantities
  - Mathematically, the floating-point number 1.0/3.0 is equal to 0.3333333 . . .
  - A computer has a finite amount of storage space
    - It may store 1.0/3.0 as something like 0.3333333333, which is slightly smaller than one-third
  - Computers actually store numbers in binary notation, but the consequences are the same: floating-point numbers may lose accuracy



- Integer and Floating-Point Division

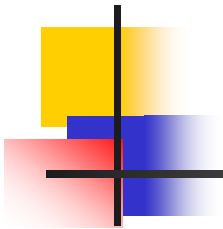
- When one or both operands are a floating-point type, division results in a floating-point type

  `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



- The % Operator

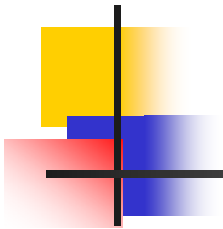
- The `%` operator is used with operands of type `int` to recover the information lost after performing integer division

  `15/2` evaluates to the quotient `7`

  `15%2` evaluates to the remainder `1`

- The `%` operator can be used to count by 2's, 3's, or any other number

  - To count by twos, perform the operation `number % 2`, and when the result is `0`, `number` is even

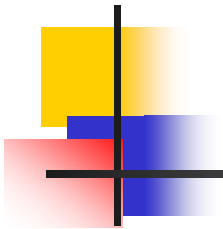


- Type Casting ...

- A *type cast* takes a value of one type and produces a value of another type with an "equivalent" value
  - If `n` and `m` are integers to be divided, and the fractional portion of the result must be preserved, at least one of the two must be type cast to a floating-point type **before** the division operation is performed

```
double ans = n / (double)m;
```

  - Note that the desired type is placed inside parentheses immediately in front of the variable to be cast
  - Note also that the type and value of the variable to be cast does not change



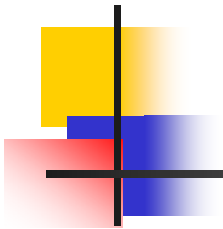
... - Type Casting

- When type casting from a floating-point to an integer type, the number is truncated, not rounded
  - `(int) 2.9` evaluates to `2`, not `3`
- When the value of an integer type is assigned to a variable of a floating-point type, Java performs an automatic type cast called a *type coercion*

```
double d = 5;
```

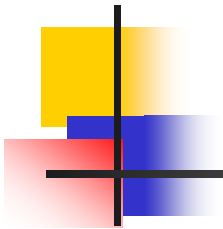
- In contrast, it is illegal to place a `double` value into an `int` variable without an explicit type cast

```
int i = 5.5; // Illegal
int i = (int)5.5 // Correct
```



- Increment and Decrement Operators ...

- The *increment operator* (**++**) adds one to the value of a variable
  - If **n** is equal to **2**, then **n++** or **++n** will change the value of **n** to **3**
- The *decrement operator* (**--**) subtracts one from the value of a variable
  - If **n** is equal to **4**, then **n--** or **--n** will change the value of **n** to **3**



... - Increment and Decrement Operators

- When either operator precedes its variable, and is part of an expression, then the expression is evaluated using the changed value of the variable
  - If **n** is equal to **2**, then **2*(++n)** evaluates to **6**
- When either operator follows its variable, and is part of an expression, then the expression is evaluated using the original value of the variable, and only then is the variable value changed
  - If **n** is equal to **2**, then **2*(n++)** evaluates to **4**

THE END