## Chapter 2 - Variables / Assignments

## Section 2.1 - Variables (int)

Here's a variation on a common schoolchild riddle.


Participation Activity

### 2.1.1: People on bus.

For each step, keep track of the current number of people by typing in the numPeople box (it's editable).

Start


| 1 | 2 | 3 | 4 | 5 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

Check
Next

You used that box to remember the number of people as you proceeded through each step. Likewise, a program uses a variable to remember values as the program executes instructions. (By the way, the real riddle's ending question is actually "What is the bus driver's name?" - the subject usually says "How should I know?". The riddler then says "I said, YOU are driving a bus.")

A variable represents a memory location used to store data. That location is like the "box" that you used above. The statement int userAge; defines (also called declares) a new variable named userAge. The compiler allocates a memory location for userAge capable of storing an integer, hence the "int". When a statement executes that assigns a value to a variable, the processor stores the value
into the variable's memory location. Likewise, reading a variable's value reads the value from the variable's memory location. ${ }^{\text {mem }}$ The animation illustrates.


In the animation, the compiler allocated variable userAge to memory location 97, known as the variables address; the choice of 97 is arbitrary, and irrelevant to the programmer (but the idea of a memory location is important to understand). The animation shows memory locations 96-99; a real memory will have thousands, millions, or even billions of locations.

Although not required, an integer variable is commonly assigned an initial value when defined.

## Construct 2.1.1: Basic integer variable definition with initial value of 0 .

```
int variableName = 0;
```



Note: Capitalization matters, so MyNumber is not the same as myNumber.

| \# | Question | Your answer |
| :--- | :--- | :--- |
| 1 | Define an integer variable named numPeople. Do <br> not initialize the variable. | $\square$ |
| 2 | Define an integer variable named numDogs, <br> initializing the variable to 0 in the definition. | $\square$ |
| 3 | Define an integer variable named daysCount, <br> initializing the variable to 365 in the definition. | $\square$ |
| 4 | What memory location (address) will a compiler <br> allocate for the variable definition: <br> int numHouses = 99; <br> If appropriate, type: Unknown | $\square$ |

The programmer must define a variable before any statement that assigns or reads the variable, so that the variable's memory location is known.

A variable definition is also commonly called a variable declaration. This material may use either term.


Define a second integer variable avgLifespan, initialized to 70. Add a statement that prints "Average lifespan is 70" (don't type 70 there; print the avgLifespan variable).
import java.util.Scanner;
import java.util.Scanner;
public class Age {
public class Age {
public static void main (String [] args) {
public static void main (String [] args) {
Scanner scnr = new Scanner(System.in);
Scanner scnr = new Scanner(System.in);
int userAge = 0;
int userAge = 0;
// Define new variable here
// Define new variable here
System.out.println("Enter your age: ");
System.out.println("Enter your age: ");
userAge = scnr.nextInt();
userAge = scnr.nextInt();
System.out.println(userAge + " is a great age.");
System.out.println(userAge + " is a great age.");
// Put new print statement here
// Put new print statement here
return;
return;
}
}
}
}

A common error is to read a variable that has not yet been assigned a value. If a local variable is defined but not initialized, the variable's memory location contains some unknown value, commonly but not always 0 . A program with an uninitialized variable may thus run correctly on system that has 0 in the memory location, but then fail on a different system - a very difficult bug to fix. Programmers thus must ensure that a program assigns a variable before reading. A good practice is to initialize a variable in its definition whenever practical. The space allocated to a variable in memory is not infinite. An int variable can usually only hold numbers in the range $-2,147,483,648$ to $2,147,483,647$. That's about $\pm 2$ billion.

| Which statement is an error? | Participation <br> Activity | 2.1.5: int variables. |  |
| :--- | :--- | :--- | :--- |
| \# | Question | Your answer |  |
|  | int dogCount; | Error |  |
| 1 |  | No error |  |
| 2 | int amountOwed $=-999 ;$ | Error |  |
| 2 | int numYears = 9000111000; | No error |  |
| 3 |  | Error |  |

Multiple variables can be defined in the same statement, as in:
int numProtons, numNeutrons, numElectrons; This material usually avoids such style, especially when definition initializes the variable (which may be harder to see otherwise).

### 2.1.1: Declaring variables.

Write one statement that declares an integer variable numHouses initialized to 25 .

```
import java.util.Scanner;
public class DeclaringVariables {
    public static void main (String [] args) {
        /* Your solution goes here */
        System.out.println(numHouses);
        return;
    }
}
```

Run
(*mem) Instructors: Although compilers may optimize variables away or store them on the stack or in a register, the conceptual view of a variable in memory helps understand many language aspects.

## Section 2.2 - Assignments

An assignment statement like numApples = 8; stores (i.e. assigns) the right-side item's current value (in this case, 8) into the variable on left side (numApples). ${ }^{\text {asgn }}$

## Construct 2.2.1: Assignment statement.

variableName = expression;

An expression may be a number like 80, a variable name like numApples, or a simple calculation like numApples +1 . Simple calculations can involve standard math operators like + , -, and *, and parentheses as in 2 * (numApples - 1). Another section describes expressions further.

Figure 2.2.1: Assigning a variable.

```
public class Mice {
        public static void main(String [] args) {
            int litterSize = 3; // Low end of litter size range
            int yearlyLitters = 5; // Low end of litters per year
            int annualMice = 0;
            System.out.print("One female mouse may give birth to ");
            annualMice = litterSize * yearlyLitters;
            System.out.println(annualMice + " mice,");
            litterSize = 14; // High end
            yearlyLitters = 10; // High end
            System.out.print("and up to ");
            annualMice = litterSize * yearlyLitters;
            System.out.println(annualMice + " mice, in a year.");
            return;
        }
}
```

All three variables are initialized, with annualMice initialized to 0 . Later, the value of litterSize * yearlyLitters ( 3 * 5 , or 15 ) is assigned to annualMice, which is then printed. Next, 14 is assigned to litterSize, and 10 to yearlyLitters, and their product (14 * 10, or 140) is
 assigned to annualMice, which is printed.


Select the correct value for $x, y$, and $z$ after the following code executes.

Start
int $x=5$;
int $y=3$;
int $z=9 ;$
x = 3;
$y=0$;
z $=5$;
$\mathrm{x}=4$;


Z is
195


Check
Next


Be sure to end assignment statements with a semicolon ;.

| \# | Question | Your answer |
| :--- | :--- | :--- |
| 1 | Write an assignment statement to assign 99 to <br> numCars. | $\square$ |
|  |  |  |


| 2 | Assign 2300 to houseSize. |  |
| :---: | :---: | :---: |
| 3 | Assign the current value of numApples to numFruit. |  |
| 4 | The current value in houseRats is 200. Then: <br> numRodents $=$ houseRats; <br> executes. You know 200 will be stored in numRodents. What is the value of houseRats after the statement executes? Valid answers: 0, 199, 200, or unknown. |  |
| 5 | Assign the result of ballCount - 3 to numitems. |  |
| 6 | dogCount is 5 . After <br> animalsTotal $=$ dogCount -3 ; <br> executes, what is the value in animalsTotal? |  |
| 7 | dogCount is 5 . After animalsTotal = dogCount - 3; <br> executes, what is the value in dogCount? |  |
| 8 | What is the value of numBooks after both statements execute? <br> numbooks $=5 ;$ numbooks $=3 ;$ <br> numbooks $=3$; |  |

A common error among new programmers is to assume = means equals, as in mathematics. In contrast, = means "compute the value on the right, and then assign that value into the variable on the left." Some languages use := instead of = to reduce confusion. Programmers sometimes speak numltems = numApples as "numltems EQUALS numApples", but this material strives to avoid such inaccurate wording.

Another common error by beginning programmers is to write an assignment statement in reverse, as in: numKids + numAdults = numPeople, or $9=$ beansCount. Those statements won't compile. But, writing numCats = numDogs in reverse will compile, leading to a hard-to-find bug.

Commonly, a variable appears on both the right and left side of the = operator. If numltems is initially 5 , then after numItems $=$ numItems +1 , numltems will be 6 . The statement reads the value of numltems (5), adds 1, and stores the result of 6 in numltems-overwriting whatever value was previously in numltems.

Participation Activity
2.2.3: Assigning to a variable overwrites its previous values: People-known example.

## Start

```
import java.util.Scanner;
public class PeopleKnown {
    public static void main (String [] args) {
        int yourFriends = 0;
        int totalFriends = 0;
        System.out.print("Enter the number of people you know: ");
        Scanner scnr = new Scanner(System.in);
        yourFriends = scnr.nextInt();
        totalFriends = yourFriends;
        System.out.println(" You know " + totalFriends + " people.");
        totalFriends = totalFriends * yourFriends;
        System.out.println(" Those people know " + totalFriends + " people.");
        totalFriends = totalFriends * yourFriends;
        System.out.println(" And they know " + totalFriends + " people.");
        return;
    }
}
```

Enter the number of people you know: 200
You know 200 people.
Those people know 40000 people.
And they know 8000000 people.

(The above example relates to the popular idea that any two people on earth are connected by just "six degrees of separation", accounting for overlapping of known-people.

| Participation | 2.2.4: Assignment statements with same variable on both sides. |  |
| :--- | :--- | :--- |
| Activity | Question | Your answer |
| 1 | numApples is initially 5. What is numApples after: <br> numApples $=$ numApples $+3 ;$ | $\square$ |
| 2 | numApples is initially 5. What is numFruit after: <br> numFruit $=$ numApples ; <br> numFruit $=$ numFruit $+1 ;$ | $\square$ |
| 3 | Write a statement ending with - 1 that decreases <br> variable flyCount's value by 1. | $\square$ |



Give the final value of $z$ after the statements execute.

| \# | Question | Your answer |
| :---: | :---: | :---: |
| 1 | $\begin{aligned} & \mathrm{w}=1 ; \\ & \mathrm{y}=2 ; \\ & \mathrm{z}=4 ; \\ & \mathrm{x}=\mathrm{y}+1 ; \\ & \mathrm{w}=2-\mathrm{x} ; \\ & \mathrm{z}=\mathrm{w} * \mathrm{y} ; \end{aligned}$ |  |
| 2 | $\begin{aligned} & \mathrm{x}=4 ; \\ & \mathrm{y}=0 ; \\ & \mathrm{z}=3 ; \end{aligned}$ $\begin{aligned} & x=x-3 ; \\ & y=y+x ; \\ & z=z * y ; \end{aligned}$ |  |
| 3 | $\begin{aligned} & \mathrm{x}=6 ; \\ & \mathrm{y}=-2 ; \\ & \mathrm{y}=\mathrm{x}+\mathrm{x} ; \\ & \mathrm{w}=\mathrm{y} * \mathrm{x} ; \\ & \mathrm{z}=\mathrm{w}-\mathrm{y} ; \end{aligned}$ |  |
| 4 | $\begin{aligned} & \mathrm{w}=-2 ; \\ & \mathrm{x}=-7 ; \\ & \mathrm{y}=-8 ; \\ & \mathrm{z}=\mathrm{x}-\mathrm{y} ; \\ & \mathrm{z}=\mathrm{z} * \mathrm{w} ; \\ & \mathrm{z}=\mathrm{z} / \mathrm{w} ; \end{aligned}$ |  |

### 2.2.1: Enter the output of the variable assignments.

Start

## Enter the output of the following program.

public class combinedOutput \{
public static void main (String [] args) \{
int $x=0$;
int $y=6$;
$x=9 ;$
System.out.print(x + " " + y);

return;
\}
\}

| 1 | 2 |  | 3 | 4 | 5 |  | 6 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

Check
Next

## Challenge <br> Activity <br> 2.2.2: Assigning a value.

Write a statement that assigns 3 to hoursLeft.

```
import java.util.Scanner;
public class AssignmentValue {
    public static void main (String [] args) {
        int hoursLeft = 0;
        /* Your solution goes here */
        System.out.print(hoursLeft);
        System.out.println(" hours left.");
        return;
    }
}
```

Run

### 2.2.3: Assigning a sum.

Write a statement that assigns numNickels + numDimes to numCoins. Ex: 5 nickels and 6 dimes res

```
public class AssigningSum {
    public static void main (String [] args) {
    int numCoins = 0;
    int numNickels = 0;
    int numDimes = 0;
    numNickels = 5;
    numDimes = 6;
    /* Your solution goes here */
    System.out.print("There are ");
    System.out.print(numCoins);
    System.out.println(" coins");
    return;
    }
}
```

Run

### 2.2.4: Adding a number to a variable.

Write a statement that increases numPeople by 5 . If numPeople is initially 10 , then numPeople becor

```
import java.util.Scanner;
public class AssigningNumberToVariable {
    public static void main (String [] args) {
        int numPeople = 0;
        numPeople = 10;
        /* Your solution goes here */
        System.out.print("There are ");
        System.out.print(numPeople);
        System.out.println(" people.");
        return;
    }
}
```

Run
(*asgn) We ask instructors to give us leeway to teach the idea of an "assignment statement," rather than the language's actual "assignment expression," whose use we condone primarily in a simple statement.

## Section 2.3 - Identifiers

A name created by a programmer for an item like a variable or method is called an identifier. An identifier must be a sequence of letters (a-z, A-Z, _, \$) and digits (0-9) and must start with a letter. Note that "_", called an underscore, and "\$", called a dollar sign or currency symbol, are considered to be letters. A good practice followed by many Java programmers is to not use _ or $\$$ in programmercreated identifiers.

The following are valid identifiers: c, cat, Cat, n1m1, short1, and _hello. Note that cat and Cat are different identifiers. The following are invalid identifiers: 42c (starts with a digit), hi there (has a disallowed symbol: space), and cat! (has a disallowed symbol: !).

A reserved word is a word that is part of the language, like int, short, or double. A reserved word is also known as a keyword. A programmer cannot use a reserved word as an identifier. Many language editors will automatically color a program's reserved words. A list of reserved words appears at the end of this section.

|  | Participation Activity | 2.3.1: Valid identifiers. |  |
| :---: | :---: | :---: | :---: |
| Which are valid identifiers? |  |  |  |
| \# | Question |  | Your answer |
| 1 | numCars |  | Valid |
|  |  |  | Invalid |
| 2 | num_Cars1 |  | Valid |
|  |  |  | Invalid |
| 3 | _numCars |  | Valid |
|  |  |  | Invalid |
| 4 | __numCars |  | Valid |
|  |  |  | Invalid |
| 5 | num cars |  | Valid |
|  |  |  | Invalid |


| 6 | 3rdPlace | Valid |  |
| :--- | :--- | :--- | :--- |
| 7 |  | Invalid |  |
| 8 | thirdPlace_ | Valid |  |
| 9 | thirdPlace! | Invalid |  |
| tall | Valid |  |  |
| 10 |  | Invalid |  |
|  |  | Valid |  |
|  |  |  | Invalid |
|  | short | Valid |  |
|  |  | Invalid |  |



Note: Doesn't consider library items.
$\square$

Awaiting your input...

Identifiers are case sensitive, meaning upper and lower case letters differ. So numCats and NumCats are different.

While various (crazy-looking) identifiers may be valid, programmers follow identifier naming conventions (style) defined by their company, team, teacher, etc. Two common conventions for naming variables are:

- Camel case: Lower camel case abuts multiple words, capitalizing each word except the first, as in numApples or peopleOnBus.
- Underscore separated: Words are lowercase and separated by an underscore, as in num_apples or people_on_bus.

This material uses lower camel case; that style is recommend by the creators of Java in their naming conventions document. Consistent style makes code easier to read and maintain, especially if multiple programmers will be maintaining the code.

Programmers should follow the good practice of creating meaningful identifier names that selfdescribe an item's purpose. Meaningful names make programs easier to maintain. The following are fairly meaningful: userAge, houseSquareFeet, and numltemsOnShelves. The following are less meaningful: age (whose age?), sqft (what's that stand for?), num (almost no info). Good practice minimizes use of abbreviations in identifiers except for well-known ones like num in numPassengers. Abbreviations make programs harder to read and can also lead to confusion, such as if a chiropractor application involves number of messages and number of massages, and one is abbreviated numMsgs (which is it?).

This material strives to follow another good practice of using two or more words per variable such as numStudents rather than just students, to provide meaningfulness, to make variables more recognizable when they appear in writing like in this text or in a comment, and to reduce conflicts with reserved words or other already-defined identifiers.

While meaningful names are important, very long variable names, such as
averageAgeOfUclaGraduateStudent, can make subsequent statements too long and thus hard to read. Programmers strive to find a balance.

Choose the "best" identifier for a variable with the stated purpose, given the above discussion (including this material's variable naming convention).

| \# | Question | Your answer |
| :--- | :--- | :--- |
| The number of students attending UCLA. | num |  |
| 2 |  | numStdsUcla |
| The size of an LCD monitor | numStudentsUcla |  |
|  |  | numberOfStudentsAttendingUcla |
|  |  | size |
|  |  | sizeLcdMonitor |
|  |  | nelly |

## Table 2.3.1: Java reserved words / keywords.

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

See Wikipedia: Java keywords for a description of each reserved word.
The words "true", "false", and "null" are also reserved, used for literals.

## Section 2.4 - Arithmetic expressions (int)

An expression is a combination of items, like variables, literals, and operators, that evaluates to a value. An example is: 2 * (numltems + 1). If numltems is 4 , then the expression evaluates to 2 * $(4+1)$ or 10. A literal is a specific value in code like 2. Expressions occur in variable definitions and in assignment statements (among other places).

Figure 2.4.1: Example expressions in code.

```
int numKids = 0; // Expr: 0
numKids = 7; // Expr: 7
numPeople = numKids + numAdults; // Expr: numKids + numAdults
totOffers = jobsCA + (2 * jobsAZ); // Expr: jobsCA + (2 * jobsAZ)
xCoord = yCoord; // Expr: yCoord
xCoord = -yCoord; // Expr: -yCoord
```

Note that an expression can be just a literal, just a variable, or some combination of variables, literals, and operators.

Commas are not allowed in an integer literal. So 1,333,555 is written as 1333555 .

|  | Participation Activity | 2.4.1: Expression in statements. |  |
| :---: | :---: | :---: | :---: |
| \# | Question |  | Your answer |
| 1 | Is the following an expression? 12 |  | Yes |
|  |  |  | No |
| 2 | Is the following an expression? int eggsInCarton |  | Yes |
|  |  |  | No |
| 3 | Is the following an expression? eggsInCarton * 3 |  | Yes |
|  |  |  | No |
| 4 | Is the following an error? An int's maximum value is 2,147,483,647. <br> numYears $=1,999,999,999$; |  | Yes |
|  |  |  | No |

An operator is a symbol for a built-in language calculation like + for addition. Arithmetic operators built into the language are:

## Table 2.4.1: Arithmetic operators.

| Arithmetic operator | Description |
| :--- | :--- |
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| \% | modulo (remainder) |

Modulo may be unfamiliar and is discussed further below.
Parentheses may be used, as in: ((userltems + 1) * 2) / totalltems. Brackets [ ] or braces \{ \} may NOT be used.

Expressions mostly follow standard arithmetic rules, such as order of evaluation (items in parentheses first, etc.). One notable difference is that the language does not allow the multiplication shorthand of abutting a number and variable, as in $5 y$ to represent 5 times $y$.

|  | Participation <br> Activity | 2.4.2: Capturing behavior with an expressions. |
| :--- | :--- | :--- | :--- |
| Does the expression correctly capture the intended behavior? |  |  |
| $\#$ | Question | Your answer |
|  | 6 plus numiltems: | Yes |
| 1 | 6 + numItems | No |
| 2 | 6 times numitems: | Yes |
| 3 | numItems | No |
| totDays divided by 12: | Yes |  |


|  |  | Ivu |
| :--- | :--- | :--- |
| 4 | 5 times i: | Yes |
| 5 | The negative of userVal: <br> -userVal | No |
| 6 | itemsA + itemsB / 2 | Yes |
| 7 | n! | No |
|  |  | Yes |

## Figure 2.4.2: Expressions examples: Leasing cost.

```
import java.util.Scanner;
/* Computes the total cost of leasing a car given the down payment,
    monthly rate, and number of months
*/
public class CarLeaseCost \{
    public static void main (String [] args) \{
            Scanner scnr = new Scanner(System.in);
            int downpayment \(=0\);
            int paymentPerMonth \(=0\);
            int numMonths \(=0\);
            int totalCost \(=0 ; / /\) Computed total cost to be output
            System.out.println("Enter down payment: ");
            downpayment \(=\) scnr.nextInt();
            System.out.println("Enter monthly payment: ");
            paymentPerMonth = scnr.nextInt();
            System.out.println("Enter number of months: ");
            numMonths = scnr.nextInt();
            totalCost \(=\) downpayment + (paymentPerMonth * numMonths);
            System.out.println("Total cost: " + totalCost);
            return;
        \}
\}
```

A good practice is to include a single space around operators for readability, as in numitems +2 , rather than numltems+2. An exception is - used as negative, as in: xCoord $=-y$ Coord. - used as negative is known as unary minus.

|  | Participation <br> Activity | 2.4.3: Single space around operators. |
| :--- | :--- | :--- |
| Retype each statement to follow the good practice of a single space around operators. |  |  | | $\#$ | Question | Your answer |
| :--- | :--- | :--- |
| 1 | housesCity $=$ housesBlock *10; | $\square$ |
| 2 | $\mathrm{x}=\mathrm{x} 1+\mathrm{x} 2+2$; | $\square$ |
| 3 | numBalls=numBalls+1; |  |
| 4 | numEntries $=$ (userval+1)*2; | $\square$ |

When the / operands are integers, the division operator / performs integer division, throwing away any remainder. Examples:

- $24 / 10$ is 2 .
- $50 / 50$ is 1 .
- $1 / 2$ is 0.2 divides into 1 zero times; remainder of 1 is thrown away.

A common error is to forget that a fraction like (1/2) in an expression performs integer division, so the expression evaluates to 0 .

The modulo operator \% may be unfamiliar to some readers. The modulo operator evaluates to the remainder of the division of two integer operands. Examples:

- $24 \% 10$ is 4. Reason: 24 / 10 is 2 with remainder 4.
- $50 \% 50$ is 0 . Reason: $50 / 50$ is 1 with remainder 0 .
- $1 \% 2$ is 1 . Reason: $1 / 2$ is 0 with remainder 1 .


## Figure 2.4.3: Division and modulo example: Minutes to hours/minutes.

```
import java.util.Scanner;
public class TimeConverter {
        public static void main (String [] args) {
            Scanner scnr = new Scanner(System.in);
            int userMinutes = 0; // User input: Minutes
            int outHours = 0; // Output hours
            int outMinutes = 0; // Output minutes (remai Enter minutes:
            System.out.println("Enter minutes: ");
            userMinutes = scnr.nextInt();
            outHours = userMinutes / 60;
            outMinutes = userMinutes % 60;
            System.out.print(userMinutes + " minutes is
```

Enter minutes:
180
180 minutes is 3 hours and 0 mis

```
            System.out.print(outHours + " hours and ");
            System.out.println(outMinutes + " minutes.");
            return;
    }
}
```

For integer division, the second operand of / or \% must never be 0 , because division by 0 is mathematically undefined. A divide-by-zero error occurs at runtime if a divisor is 0 , causing a program to terminate.

## Figure 2.4.4: Divide-by-zero example: Compute salary per day.

```
import java.util.Scanner;
public class DailySalary {
        public static void main (String [] args) {
            Scanner scnr = new Scanner(System.in);
            int salaryPerYear = 0; // User input: Yearly salary
            int daysPerYear = 0; // User input: Days worked per year
            int salaryPerDay = 0; // Output: Salary per day
            System.out.println("Enter yearly salary:");
            salaryPerYear = scnr.nextInt();
            System.out.println("Enter days worked per year:");
            daysPerYear = scnr.nextInt();
60000
Enter days
0
Exception :
                                    at
```

Enter year:

```
            // If daysPerYear is 0, then divide-by-zero causes program termination.
            salaryPerDay = salaryPerYear / daysPerYear;
            System.out.println("Salary per day is: " + salaryPerDay);
        return;
    }
}
```



Determine the result. Type "Error" if the program would terminate due to divide-by-zero. Only literals appear in these expressions to focus attention on the operators; most practical expressions include variables.

| \# | Question | Your answer |
| :---: | :---: | :---: |
| 1 | 13 / 3 | $\square$ |
| 2 | $4 / 9$ |  |
| 3 | $(5+10+15) *(1 / 3)$ |  |
| 4 | 50 \% 2 |  |
| 5 | 51 \% 2 | $\square$ |
| 6 | 78 \% 10 |  |
| 7 | $596 \% 10$ |  |
| 8 | $100 /(1 / 2)$ | $\square$ |

The compiler evaluates an expression's arithmetic operators using the order of standard mathematics, such order known in programming as precedence rules.

## Table 2.4.2: Precedence rules for arithmetic operators.

| Convention | Description | Explanation |
| :---: | :---: | :---: |
| () | Items within parentheses are evaluated first | In 2 * ( $\mathrm{A}+1$ ), $\mathrm{A}+1$ is computed first, with the result then multiplied by 2 . |
| unary - | - used as a negative <br> (unary minus) is next | In 2 * -A, -A is computed first, with the result then multiplied by 2 . |
| * / \% | Next to be evaluated are *, /, and \%, having equal precedence. |  |
| +- | Finally come + and with equal precedence. | In B $=3+2 * A, 2$ * $A$ is evaluated first, with the result then added to 3 , because * has higher precedence than + . Note that spacing doesn't matter: $B=3+2$ * A would still evaluate 2 * $A$ first. |
| left-toright | If more than one operator of equal precedence could be evaluated, evaluation occurs left to right. | $\ln B=A * 2 / 3, A * 2$ is first evaluated, with the result then divided by 3 . |

A common error is to omit parentheses and assume an incorrect order of evaluation, leading to a bug. For example, if $x$ is $3,5 * x+1$ might appear to evaluate as $5 *(3+1)$ or 20 , but actually evaluates as (5*3) + 1 or 16 (spacing doesn't matter). Good practice is to use parentheses to make order of evaluation explicit, rather than relying on precedence rules, as in: $y=(m * x)+b$, unless order doesn't matter as in $x+y+z$.

Figure 2.4.5: Post about parentheses.
(Poster A): Tried rand() \% (35-18) + 18, but it's wrong.

11
Use these
(Poster B): I don't understand what you're doing with (35-18) +18 . Wouldn't that just be 35 ?
(Poster C): The \% operator has higher precedence than the + operator. So read that as (rand() \% (35-18)) +18 .


Participation Activity
2.4.5: Precedence rules.

Select the expression whose parentheses enforce the compiler's evaluation order for the original expression.

| \# | Question | Your answer |
| :---: | :---: | :---: |
| 1 | $y+2^{*} z$ | $(y+2)$ * $z$ |
|  |  | $y+\left(2^{*} z\right)$ |
| 2 | z/2-x | (z/2) - x |
|  |  | z / (2-x) |
| 3 | $x^{*} y^{*} z$ | $\left(x^{*} y\right)^{*} z$ |
|  |  | $x^{*}\left(y^{*} z\right)$ |
| 4 | $x+y$ \% 3 | $(x+y) \% 3$ |
|  |  | $x+\left(\begin{array}{l}\text { \% 3 3 }\end{array}\right.$ |
| 5 | $x+1^{*} y / 2$ | $\left((x+1)^{*} y\right) / 2$ |
|  |  | $x+\left(\left(1^{*} y\right) / 2\right)$ |
|  |  | $x+(1 *$ (y/2) $)$ |
| 6 | $x / 2+y / 2$ | $((x / 2)+y) / 2$ |
|  |  | $(\mathrm{x} / 2)+(y / 2)$ |

What is totCount after executing the following?
numItems = 5;
7

44

The above question set helps make clear why using parentheses to make order of evaluation explicit is good practice. (It also intentionally violated spacing guidelines to help make the point).

Special operators called compound operators provide a shorthand way to update a variable, such as userAge $+=1$ being shorthand for userAge $=$ userAge +1 . Other compound operators include -=, *=, /=, and \%=.


If appropriate, type: Not possible

| \# | Question | Your answer |
| :--- | :--- | :---: |
| 1 | numAtoms is initially 7. What is numAtoms after: <br> numAtoms += 5? | $\square$ |
| 2 | numAtoms is initially 7. What is numAtoms after: <br> numAtoms *= 2? | $\square$ |
| 3 | Rewrite the statement using a compound operator: <br> carCount = carCount / 2; | $\square$ |
| 4 | Rewrite the statement using a compound operator: <br> numltems = boxCount + 1; | $\square$ |

### 2.4.1: Enter the output of the integer expressions.

Start

## Enter the output of the following program.

public class combinedOutput \{
public static void main (String [] args) \{
int $x=2$;
int $y=0$;
$y=3 *(x+8) ;$
System.out.print(x + " " + y);
230
return;
\}
\}

| 1 | 2 | 3 | 4 | 5 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

Check
Next

### 2.4.2: Compute an expression.

Write a statement that computes num1 plus num2, divides by 3, and assigns the result to finalResult result is 3 .

```
lmport Java.utll.Scanner;
public class ComputingFinalResult {
    public static void main (String [] args) {
        int num1 = 0;
        int num2 = 0;
        int finalResult = 0;
        num1 = 4;
        num2 = 5;
        /* Your solution goes here */
        System.out.print("Final result: ");
        System.out.println(finalResult);
        return;
    }
}
```

Run

### 2.4.3: Compute change.

A cashier distributes change using the maximum number of five dollar bills, followed by one dollar bill 4 ones. Write a single statement that assigns the number of one dollar bills to variable numOnes, give operator.

```
pubLlc class ComputrngChange {
    public static void main (String [] args) {
        int amountToChange = 0;
        int numFives = 0;
        int numOnes = 0;
        amountToChange = 19;
        numFives = amountToChange / 5;
        /* Your solution goes here */
        System.out.print("numFives: ");
        System.out.println(numFives);
        System.out.print("numOnes: ");
        System.out.println(numOnes);
        return;
    }
}
```

Run

### 2.4.4: Total cost.

A drink costs 2 dollars. A taco costs 3 dollars. Given the number of each, compute total cost and as: tacos yields totalCost of 26 .

```
lmport Java.util.Scanner;
public class ComputingTotalCost {
    public static void main (String [] args) {
        int numDrinks = 0;
        int numTacos = 0;
        int totalCost = 0;
        numDrinks = 4;
        numTacos = 6;
        /* Your solution goes here */
        System.out.print("Total cost: ");
        System.out.println(totalCost);
        return;
    }
}
```

Run

## Section 2.5 - Floating-point numbers (double)

A variable is sometimes needed to store a floating-point number like -1.05 or 0.001 . A variable defined as type double stores a floating-point number.

> Construct 2.5.1: Floating-point variable definition with initial value of 0.0.
> double variableName $=0.0 ; / /$ Initial value is optional but recommended.

A floating-point literal is a number with a fractional part, even if that fraction is 0 , as in $1.0,0.0$, or
99.573. Good practice is to always have a digit before the decimal point, as in 0.5 , since .5 might mistakenly be viewed as 5 ..

## Figure 2.5.1: Variables of type double: Travel time example.

```
import java.util.Scanner;
public class TravelTime {
    public static void main (String [] args) {
            Scanner scnr = new Scanner(System.in);
            double milesTravel = 0.0; // User input of miles to travel
            double hoursFly = 0.0; // Travel hours if flying those m
            double hoursDrive = 0.0; // Travel hours if driving those
            System.out.print("Enter a distance in miles:\n");
            milesTravel = scnr.nextDouble();
            hoursFly = milesTravel / 500.0;
            hoursDrive = milesTravel / 60.0;
            System.out.println(milesTravel + " miles would take:");
            System.out.println(hoursFly + " hours to fly,");
            System.out.println(hoursDrive + " hours to drive.");
            return;
    }
}
```

Note that reading a floating-point value from input uses nextDouble(), in contrast to using nextInt) to read an integer.

$\square$| Participation <br> Activity |
| :--- |

All variables are of type double and already-defined unless otherwise noted.

| \# | Question | Your answer |
| :--- | :--- | :--- |
| 1 | Define a double variable named personHeight and <br> initialize to 0.0. | $\square$ |
| 2 | Compute ballHeight divided by 2.0 and assign the <br> result to ballRadius. Do not use the fraction 1.0 / <br> 2.0; instead, divide ballHeight directly by 2.0. | $\square$ |
| 3 | Multiply ballHeight by the fraction one half, namely <br> (1.0 / 2.0), and assign the result to ballRadius. Use <br> the parentheses around the fraction. | $\square$ |


|  | Participation Activity | 2.5.2: Floating-point literals. |  |
| :---: | :---: | :---: | :---: |
| \# | Question |  | Your answer |
| 1 | Which statement best defines and initializes the double variable? |  | double currHumidity = 99\%; |
|  |  |  | $\begin{aligned} & \text { double currHumidity = } \\ & \text { 99.0; } \end{aligned}$ |
|  |  |  | ```double currHumidity = 99;``` |
| 2 | Which statement best assigns to the variable? Both variables are of type double. |  | cityRainfall = measuredRain - 5; |
|  |  |  | cityRainfall = measuredRain - 5.0; |
| 3 | Which statement best assigns to the variable? cityRainfall is of type double. |  | cityRainfall = .97; |
|  |  |  | cityRainfall $=0.97$; |

Scientific notation is useful for representing floating-point numbers that are much greater than or much less than 0 , such as $6.02 \times 10^{23}$. A floating-point literal using scientific notation is written using an e preceding the power-of-10 exponent, as in 6.02 e 23 to represent $6.02 \times 10^{23}$. The e stands for exponent. Likewise, 0.001 is $1 \times 10^{-3}$ so 0.001 can be written as $1.0 \mathrm{e}-3$. For a floating-point literal, good practice is to make the leading digit non-zero.


In general, a floating-point variable should be used to represent a quantity that is measured, such as a distance, temperature, volume, weight, etc., whereas an integer variable should be used to represent a quantity that is counted, such as a number of cars, students, cities, minutes, etc. Floating-point is also used when dealing with fractions of countable items, such as the average number of cars per household. Note: Some programmers warn against using floating-point for money, as in 14.53 representing 14 dollars and 53 cents, because money is a countable item (reasons are discussed further in another section). int may be used to represent cents, or to represent dollars when cents are not included as for an annual salary (e.g., 40000 dollars, which are countable).


Choose the right type for a variable to represent each item.

| \# | Question | Your answer |
| :--- | :--- | :--- |
| 1 | The number of cars in a parking lot. | double |
| 2 | The current temperature in Celsius. | int |
| 3 | A person's height in centimeters. | double |
| 4 | The number of hairs on a person's head. | int |
| 5 |  | int |
|  | The average number of kids per household. | double |
|  |  | int |
|  |  | int |

A floating-point divide-by-zero occurs at runtime if a divisor is 0.0. Dividing by zero results in inf or inf depending on the signs of the operands.


### 2.5.1: Sphere volume.

Given sphereRadius and piVal, compute the volume of a sphere and assign to sphereVolume. Use (4 division, instead of (4/3) which performs integer division.

Volume of sphere $=(4.0 / 3.0) \pi r^{3}$

```
public class SphereVolumeCalculator {
        public static void main (String [] args) {
            final double piVal = 3.14159;
            double sphereVolume = 0.0;
            double sphereRadius = 0.0;
            sphereRadius = 1.0;
            /* Your solution goes here */
            System.out.println("Sphere volume: " + sphereVolume);
            return;
    }
}
```

Run

## Challenge Activity

### 2.5.2: Acceleration of gravity.

Compute the acceleration of gravity for a given distance from the earth's center, distCenter, assigninc expression for the acceleration of gravity is: $(G * M) /\left(d^{2}\right)$, where $G$ is the gravitational constant 6.673 $5.98 \times 10^{24}(\mathrm{in} \mathrm{kg})$ and d is the distance in meters from the earth's center (stored in variable distCent

```
public class GravityCalculation {
    public static void main (String [] args) {
        final double G = 6.673e-11;
        final double M = 5.98e24;
        double accelGravity = 0.0;
        double distCenter = 0.0;
        distCenter = 6.38e6;
        /* Your solution goes here */
        System.out.println("accelGravity: " + accelGravity);
        return;
    }
}
```

Run

## Section 2.6 - Constant variables

A good practice is to minimize the use of literal numbers in code. One reason is to improve code readability. newPrice $=$ origPrice -5 is less clear than newPrice $=$ origPrice - priceDiscount. When a variable represents a literal, the variable's value should not be changed in the code. If the programmer precedes the variable definition with the keyword final, then the compiler will report an error if a later statement tries to change that variable's value. An initialized variable whose value cannot change is called a constant variable. A constant variable is also known as a final variable. A common convention, or good practice, is to name constant variables using upper case letters with words separated by underscores, to make constant variables clearly visible in code.

## Figure 2.6.1: Final variable example: Lightning distance.

```
import java.util.Scanner;
// Estimates distance of lightning based on seconds
// between lightning and thunder
public class LightningDist \{
    public static void main (String[] args) \{
            Scanner scnr = new Scanner(System.in);
            final double SPEED_OF_SOUND = 761.207; // Miles/hour (sea level)
            final double SECOND̄__्रER_HOUR = 3600.0; // Secs/hour
            double secondsBetween \(=\overline{0} .0\);
            double timeInHours \(=0.0\);
            double distInMiles \(=0.0\);
            System.out.println("Enter seconds between");
            System.out.println("lightning strike and thunder:");
            secondsBetween = scnr.nextDouble();
            timeInHours = secondsBetween / SECONDS_PER_HOUR;
            distInMiles \(=\) SPEED_OF_SOUND * timeInHours;
                                    Enter sec
                                    lightning
                                    7
                                    Lightning
                                    1.4801247:
                                    . . .
                                    Enter sec
                                    lightning
                                    1
                                    Lightning
                                    0.2114463 i
```

```
            System.out. println("Lightning strike was approximately");
            System.out.println(distInMiles + " miles away.");
            return;
    \}
\}
```

Participation Activity
2.6.1: Constant variables.

Which of the following statements are valid definitions and uses of a constant integer variable named STEP_SIZE?

| \# | Question | Your answer |
| :---: | :---: | :---: |
| 1 | int STEP_SIZE = 5; | True |
|  |  | False |
| 2 | final int STEP_SIZE = 14; | True |
|  |  | False |
| 3 | totalStepHeight = numSteps * STEP_SIZE; | True |
|  |  | False |
| 4 | STEP_SIZE = STEP_SIZE + 1; | True |
|  |  | False |

## Challenge Activity

### 2.6.1: Using constants in expressions.

Assign shipCostCents with the cost of shipping a package weighing shipWeightPounds. The cost to cents plus 25 cents per pound. Declare and use a final int named CENTS_PER_POUND.

```
import java.util.Scanner;
public class ShippingCalculator {
    public static void main (String [] args) {
        int shipWeightPounds = 10;
        int shipCostCents = 0;
        final int FLAT_FEE_CENTS = 75;
        /* Your solution goes here */
        System.out.print("Weight(lb): " + shipWeightPounds);
        System.out.print(", Flat fee(cents): " + FLAT_FEE_CENTS);
        System.out.print(", Cents per pound: " + CENTS_PER_POUND);
        System.out.println(", Shipping cost(cents): " + shipCostCents);
        return;
    }
}
```

Run

## Section 2.7 - Using math methods

Some programs require math operations beyond basic operations like + and *, such as computing a square root or raising a number to a power. Thus, Java comes with a standard Math class that has about 30 math operations available, listed later in this section. As shown below, the programmer first imports the class at the top of a file (highlighted yellow), and then can use math operations (highlighted orange).

```
Figure 2.7.1: Using a math method from the math class.
import java.lang.Math;
...
double sideSquare = 0.0;
double areaSquare = 49.0;
sideSquare = Math.sqrt(areaSquare);
```

sqrt is a method. A method is a list of statements that can be executed by referring to the method's name. An input value to a method appears between parentheses and is known as an argument, such as areaSquare above. The method executes and returns a new value. In the example above, Math.sqrt(areaSquare) returns 7.0, which is assigned to sideSquare. Invoking a method is a method call.

Some methods have multiple arguments. For example, Math.pow(b, e) returns the value of $b^{e}$.

## Figure 2.7.2: Math method example: Mass growth.

```
import java.util.Scanner;
import java.lang.Math;
public class MassGrowth {
        public static void main(String[] args) {
            Scanner scnr = new Scanner(System.in);
            double initMass = 0.0; // Initial mass of a substance
            double growthRate = 0.0; // Annual growth rate
            double yearsGrow = 0.0; // Years of growth
            double finalmass = 0.0; // Final mass after those years
            System.out.print("Enter initial mass: ");
            initMass = scnr.nextDouble();
            System.out.print("Enter growth rate (Ex: 0.05 is 5%/year): ");
            growthRate = scnr.nextDouble();
            System.out.print("Enter years of growth: ");
            yearsGrow = scnr.nextDouble();
            finalMass = initMass * Math.pow(1.0 + growthRate, yearsGrow);
            // Ex: Rate of 0.05 yields initMass * 1.05^yearsGrow
            System.out.print(" Final mass after ");
            System.out.print(yearsGrow);
            System.out.print(" years is: ");
            System.out.println(finalMass);
        return;
    }
}
Enter initial mass: 10000
Enter growth rate (Ex: 0.05 is 5%/year): 0.06
Enter years of growth: 20
    Final mass after 20.0 years is: 32071.35472212848
...
Enter initial mass: 10000
Enter growth rate (Ex: 0.05 is 5%/year): 0.4
Enter years of growth: 10
    Final mass after 10.0 years is: 289254.6549759998
```



### 2.7.1: Calculate Pythagorean theorem.

Select the three statements that calculate the value of $x$ in the following:

- $x=$ square-root-of( $\left(y^{2}+z^{2}\right)$


| \# | Question | Your answer |
| :---: | :---: | :---: |
| 1 | First statement is: | $\begin{aligned} & \text { temp1 = Math.pow(x, } \\ & \text { 2.0); } \end{aligned}$ |
|  |  | $\begin{aligned} & \text { temp1 = Math.pow(z , } \\ & \text { 3.0); } \end{aligned}$ |
|  |  | $\begin{aligned} & \text { temp1 = Math.pow(y , } \\ & \text { 2.0); } \end{aligned}$ |
|  |  | temp1 = Math.sqrt(y); |
| 2 | Second statement is: | $\begin{aligned} & \text { temp2 = Math.sqrt(x , } \\ & 2.0) ; \end{aligned}$ |
|  |  | $\begin{aligned} & \text { temp2 = Math.pow(z , } \\ & \text { 2.0); } \end{aligned}$ |
|  |  | temp2 = Math.pow(z); |
|  |  | temp2 = x + <br> Math.sqrt(temp1 + temp2); |
| 3 | Third statement is: | ```temp2 = Math.sqrt(temp1 + temp2);``` |
|  |  | $\begin{aligned} & \text { x = Math.pow(temp1 + } \\ & \text { temp2, 2.0); } \end{aligned}$ |
|  |  | $\begin{aligned} & \text { X = Math.sqrt(temp1) } \\ & \text { + temp2; } \end{aligned}$ |
|  |  | $\begin{aligned} & \text { x = Math.sqrt(temp1 + } \\ & \text { temp2); } \end{aligned}$ |

Table 2.7.1: Some methods in the Java math class.

| Function | Description |  | Function | Description |
| :--- | :--- | :--- | :--- | :--- |


| pow | Raise to power | cos | Cosine |
| :---: | :---: | :---: | :---: |
| sqrt | Square root | sin | Sine |
| cbrt | Cube root | tan | Tangent |
| exp | Exponential function | acos | Arc cosine |
| $\log$ | Natural logarithm | asin | Arc sine |
| $\log 10$ | Common logarithm | atan | Arc tangent |
| $\log 1 p$ | Natural logarithm of value plus 1 | atan2 | Arc tangent with two parameters |
|  |  | cosh | Hyperbolic cosine |
| abs | Absolute value | sinh | Hyperbolic sine |
| ceil | Round up value | tanh | Hyperbolic tangent |
| floor | Round down value |  |  |
| round | Round to nearest integer | copySign | Copy sign from one value to another |
| max | Maximum of two values | getExponent | Returns exponent of floating-point value |
| min | Minimum of two values | IEEERemainder | Remainder of floatingpoint division |
|  |  | nextAfter | Next larger (or smaller) floating-point value |
| random | Generates random value between 0.0 and 1.0 | nextUp | Next larger floatingpoint value |
|  |  | rint | Rounds floating-point value to closest integer |
| toDegrees | Converts radians to degrees | scalb | Scales a value by a factor of two |
| toRadians | Converts degrees to radians | signum | Sign of value |
|  |  | ulp | Difference between floating-point value and next larger value |

See http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html for details.

- | Participation <br> Activity | 2.7.2: Variable assignments with math functions. |
| :--- | :--- |

Determine the final value of $z$ for the following code segments. All variables are of type double. Answer in the form 4.0.

| \# | Question | Your answer |
| :---: | :---: | :---: |
| 1 | $\begin{aligned} & y=2.3 ; \\ & z=3.5 ; \\ & z=\operatorname{Math} \cdot \operatorname{ceil}(y) ; \end{aligned}$ |  |
| 2 | $\begin{aligned} & y=2.3 ; \\ & z=3.5 ; \\ & z=\operatorname{Math} \cdot f \operatorname{loor}(z) ; \end{aligned}$ |  |
| 3 | $\begin{aligned} & y=3.7 ; \\ & z=4.5 ; \\ & z=\operatorname{Math} \cdot \operatorname{pow}(\operatorname{Math} . f l o o r(z), 2.0) ; \end{aligned}$ |  |
| 4 | $\begin{aligned} & z=15.75 ; \\ & z=\operatorname{Math} \cdot \text { Sqrt(Math.ceil(z)) } \end{aligned}$ |  |
| 5 | $\mathrm{z} \mathrm{=} \mathrm{Math.abs(-1.8);}$ |  |

### 2.7.1: Coordinate geometry.

Determine the distance between point ( $\mathrm{x} 1, \mathrm{y} 1$ ) and point ( $\mathrm{x} 2, \mathrm{y} 2$ ), and assign the result to pointsDista Distance $=$ SquareRootOf( $\left.(x 2-x 1)^{2}+(y 2-y 1)^{2}\right)$
You may declare additional variables.
Ex: For points (1.0, 2.0) and (1.0, 5.0), pointsDistance is 3.0.

```
lmport Java.utul.Scanner;
import java.lang.Math;
public class CoordinateGeometry {
    public static void main(String [] args) {
        double x1 = 1.0;
        double y1 = 2.0;
        double x2 = 1.0;
        double y2 = 5.0;
        double pointsDistance = 0.0;
        /* Your solution goes here */
        System.out.print("Points distance: ");
        System.out.println(pointsDistance);
        return;
    }
}
```

Run

### 2.7.2: Tree Height.

Simple geometry can compute the height of an object from the object's shadow length and shadow $\tan ($ angleElevation $)=$ treeHeight / shadowLength. Given the shadow length and angle of elevation, cc

```
lmport Java.Lang.Math;
public class TreeHeight {
    public static void main(String [] args) {
        double treeHeight = 0.0;
        double shadowLength = 0.0;
        double angleElevation = 0.0;
        angleElevation = 0.11693706; // 0.11693706 radians = 6.7 degrees
        shadowLength = 17.5;
        /* Your solution goes here */
        System.out.print("Tree height: ");
        System.out.println(treeHeight);
        return;
    }
}
```

Run

## Section 2.8 - Type conversions

A calculation sometimes must mix integer and floating-point numbers. For example, given that about $50.4 \%$ of human births are males, then 0.504 * numbirths calculates the number of expected males in numBirths births. If numBirths is an int variable (int because the number of births is countable), then the expression combines a floating-point and integer.

A type conversion is a conversion of one data type to another, such as an int to a double. The compiler automatically performs several common conversions between int and double types, such automatic conversion known as implicit conversion.

- For an arithmetic operator like + or *, if either operand is a double, the other is automatically converted to double, and then a floating-point operation is performed.
- For assignment =, the right side type is converted to the left side type if the conversion is possible without loss of precision.
int-to-double conversion is straightforward: 25 becomes 25.0.
double-to-int conversion may lose precision, so is not automatic.
Consider the expression 0.504 * numBirths, where numBirths is an int variable. If numBirths is 316, the compiler sees "double * int" so automatically converts 316 to 316.0 , then computes 0.504 * 316.0 yielding 159.264.


Participation Activity
2.8.1: Implicit conversions among double and int.

Type the value of the given expression, given int numltems $=5$, and double itemWeight $=0.5$. For any floating-point answer, give answer to tenths, e.g., 8.0, 6.5, or 0.1.

| $\#$ | Question | Your answer |
| :--- | :--- | :--- |
| 1 | $3.0 / 1.5$ | $\square$ |
| 2 | $3.0 / 2$ | $\square$ |
| 3 | (numltems +10 ) / 2 | $\square$ |
| 4 | (numltems +10 ) $/ 2.0$ | $\square$ |

Because of implicit conversion, statements like double someDoubleVar = 0 ; or someDoubleVar $=5$; are allowed, but discouraged. Using 0.0 or 5.0 is preferable.

Sometimes a programmers needs to explicitly convert an item's type. The following code undesirably performs integer division rather than floating-point division.

## Figure 2.8.1: Code that undesirably performs integer division.

```
import java.util.Scanner;
public class KidsPerFamilyBad {
    public static void main (String [] args) {
        int kidsInFamily1 = 3; // Should be int, not double
        int kidsInFamily2 = 4; // (know anyone with 2.3 kids?)
        int numFamilies = 2; // Should be int, not double
        double avgKidsPerFamily = 0.0; // Expect fraction, so double
        Average kids p;
            avgKidsPerFamily = (kidsInFamily1 + kidsInFamily2) / numFamilies;
            // Should be 3.5, but is 3 instead
            System.out.println("Average kids per family: " + avgKidsPerFamily);
        return;
    }
}
```

A common error is to accidentally perform integer division when floating-point division was intended.

A programmer can precede an expression with (type)expression to convert the expression's value to the indicated type. For example, if mylntVar is 7 , then (double) myIntVar converts int 7 to double 7.0. The following converts the numerator and denominator each to double to obtain floating-point division (actually, converting only one would have worked).

Such explicit conversion by the programmer of one type to another is known as type casting.

Figure 2.8.2: Using type casting to obtain floating-point division.

```
import java.util.Scanner;
public class KidsPerFamily {
    public static void main (String [] args) {
        int kidsInFamily1 = 3; // Should be int, not double
        int kidsInFamily2 = 4; // (know anyone with 2.3 kids?)
        int numFamilies = 2; // Should be int, not double
        double avgKidsPerFamily = 0.0; // Expect fraction, so double
        avgKidsPerFamily = (double)(kidsInFamily1 + kidsInFamily2)
                            / (double)numFamilies;
        System.out.println("Average kids per family: " + avgKidsPerFamily);
        return;
    }
}
```

A common error is to cast the entire result of integer division, rather than the operands, thus not obtaining the desired floating-point division. For example, (double) ( $5+10$ ) / 2 ) yields 7.0 (integer division yields 7 , then converted to 7.0) rather than 7.5.

A common type cast converts a double to an int. Ex: mylnt = (int)myDouble. The fractional part is truncated. Ex: 9.5 becomes 9.

| Participation <br> Activity | 2.8.2: Type casting. |  |
| :--- | :--- | :--- | :--- |
| \# | Question | Your answer |
|  | Which yields 2.5? | (int)(10) / (int)(4) |

## Challenge Activity

### 2.8.1: Type casting: Computing average kids per family

Compute the average kids per family. Note that the integers should be type cast to doubles.

```
import java.util.Scanner;
public class TypeCasting {
    public static void main (String [] args) {
        int numKidsA = 1;
        int numKidsB = 4;
        int numKidsC = 5;
        int numFamilies = 3;
        double avgKids = 0.0;
        /* Your solution goes here */
        System.out.print("Average kids per family: ");
        System.out.println(avgKids);
        return;
    }
}
```

Run

## Section 2.9 - Binary

Normally, a programmer can think in terms of base ten numbers. However, a compiler must allocate some finite quantity of bits (e.g., 32 bits) for a variable, and that quantity of bits limits the range of numbers that the variable can represent. Thus, some background on how the quantity of bits influences a variable's number range is helpful.

Because each memory location is composed of bits (0s and 1s), a processor stores a number using base 2, known as a binary number.

For a number in the more familiar base 10, known as a decimal number, each digit must be 0-9 and each digit's place is weighed by increasing powers of 10.

## Table 2.9.1: Decimal numbers use weighed powers of 10 .

| Decimal number with 3 digits | Representation |
| :--- | :--- |
| 212 | $2 * 10^{2}+1 * 10^{1}+2 * 10^{0}=$ <br> $2 * 100+1 * 10+2 * 1=$ <br> $200+10+2=$ <br> 212$=$ |

In base 2, each digit must be 0-1 and each digit's place is weighed by increasing powers of 2 .

## Table 2.9.2: Binary numbers use weighed powers of 2.

| Binary number with 4 bits | Representation |
| :--- | :--- |
| 1101 | $1 * 2^{3}+1 * 2^{2}+0 * 2^{1}+1 * 2^{0}=$ <br> $1 * 8+1 * 4+0 * 2+1 * 1=$ <br> 8 <br> $13+4+0+1=$ <br> 13 |

The compiler translates decimal numbers into binary numbers before storing the number into a memory location. The compiler would convert the decimal number 212 to the binary number 11010100, meaning $1^{*} 128+1^{*} 64+0 * 32+1^{*} 16+0 * 8+1^{*} 4+0 * 2+0 * 1=212$, and then store that binary number in memory.

## P <br> Participation Activity <br> 2.9.1: Understanding binary numbers.

Set each binary digit for the unsigned binary number below to 1 or 0 to obtain the decimal equivalents of 9 , then 50 , then 212 , then 255 . Note also that 255 is the largest integer that the 8 bits can represent.

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | (decimal value) |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |  |  |
| $2^{7}$ | $2^{6}$ | $2^{5}$ | $2^{4}$ | $2^{3}$ | $2^{2}$ | $2^{1}$ | $2^{0}$ |  |  |



Participation
Activity
2.9.2: Binary numbers.

| $\#$ | Question | Your answer |
| :--- | :--- | :--- |
| 1 | Convert the binary number 00001111 to a decimal <br> number. | $\square$ |
| 2 | Convert the binary number 10001000 to a decimal <br> number. | $\square$ |
| 3 | Convert the decimal number 17 to an 8-bit binary <br> number. | $\square$ |
| 4 | Convert the decimal number 51 to an 8-bit binary <br> number. | $\square$ |

## Section 2.10 - Characters

A variable of char type can store a single character, like the letter mor the symbol \%. A character literal is surrounded with single quotes, as in ' $m$ ' or '\%'.

## Figure 2.10.1: Simple char example: Arrow.

```
public class CharArrow {
    public static void main (String [] args) {
        char arrowBody = '-';
        char arrowHead = '>';
        System.out.println(arrowHead);
        System.out.println("" + arrowBody + arrowBody + arrowBody + arrowHead >
        arrowBody = 'o';
        --->
        000>
        System.out.println("" + arrowBody + arrowBody + arrowBody + arrowHead);
        return;
    }
}
```

Printing a single character variable is achieved by providing the variable name to as input System.out.print() or System.out.println(), as in System.out. println(arrowHead) ; To print multiple character variables using a single print statement, the input should start with " " + and each character variable should be separated by a + . For example, the second print statement in the above example prints "--->". The "" part of the statement ensures the input to println() is a string. Otherwise, the Java compiler will add the characters' values together and print the resulting value.

A common error is to use double quotes rather than single quotes around a character literal, as in myChar = "x", yielding a compiler error. Similarly, a common error is to forget the quotes around a character literal, as in myChar $=\mathrm{x}$, usually yielding a compiler error.


## Participation Activity

### 2.10.2: char variables.

Modify the program to use a char variable alertSym for the ! symbols surrounding the word WARNING, and test. Then, modify further to have the user input that symbol.


Under the hood, a char variable stores a number. For example, the letter m is stored as 109. A table showing the standard number used for common characters appears at this section's end. Though stored as a number, the compiler knows to output a char type as the corresponding character.


ASCII is an early standard for encoding characters as numbers. The following table shows the ASCII encoding as a decimal number (Dec) for common printable characters (for readers who have studied binary numbers, the table shows the binary encoding also). Other characters such as control characters (e.g., a "line feed" character) or extended characters (e.g., the letter "n" with a tilde above it as used in Spanish) are not shown. Sources: Wikipedia: ASCII, http://www.asciitable.com/.

Many earlier programming languages like C or C++ use ASCII. Java uses a more recent standard called Unicode. ASCII can represent 255 items, whereas Unicode can represent over 64,000 items; Unicode can represent characters from many different human languages, many symbols, and more. (For those who have studied binary: ASCII uses 8 bits, while Unicode uses 16, hence the 255 versus 64,000). Unicode's first several hundred items are the same as ASCII.The Unicode encoding for these characters has Os on the left to yield 16 bits.
ıade L. IU. I: Unaracter encoaings as numbers in the A১Cil stanaara.

| Binary | Dec | Char | Binary | Dec | Char | Binary | Dec | Char |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0100000 | 32 | space | 1000000 | 64 | @ | 1100000 | 96 |  |
| 0100001 | 33 | ! | 1000001 | 65 | A | 1100001 | 97 | a |
| 0100010 | 34 | " | 1000010 | 66 | B | 1100010 | 98 | b |
| 0100011 | 35 | \# | 1000011 | 67 | C | 1100011 | 99 | c |
| 0100100 | 36 | \$ | 1000100 | 68 | D | 1100100 | 100 | d |
| 0100101 | 37 | \% | 1000101 | 69 | E | 1100101 | 101 | e |
| 0100110 | 38 | \& | 1000110 | 70 | F | 1100110 | 102 | $f$ |
| 0100111 | 39 | ' | 1000111 | 71 | G | 1100111 | 103 | g |
| 0101000 | 40 | $($ | 1001000 | 72 | H | 1101000 | 104 | h |
| 0101001 | 41 | ) | 1001001 | 73 | 1 | 1101001 | 105 | i |
| 0101010 | 42 | * | 1001010 | 74 | $J$ | 1101010 | 106 | j |
| 0101011 | 43 | + | 1001011 | 75 | K | 1101011 | 107 | k |
| 0101100 | 44 | , | 1001100 | 76 | L | 1101100 | 108 | 1 |
| 0101101 | 45 | - | 1001101 | 77 | M | 1101101 | 109 | m |
| 0101110 | 46 | . | 1001110 | 78 | N | 1101110 | 110 | n |
| 0101111 | 47 | 1 | 1001111 | 79 | 0 | 1101111 | 111 | 0 |
| 0110000 | 48 | 0 | 1010000 | 80 | P | 1110000 | 112 | p |
| 0110001 | 49 | 1 | 1010001 | 81 | Q | 1110001 | 113 | q |
| 0110010 | 50 | 2 | 1010010 | 82 | R | 1110010 | 114 | $r$ |
| 0110011 | 51 | 3 | 1010011 | 83 | S | 1110011 | 115 | S |
| 0110100 | 52 | 4 | 1010100 | 84 | T | 1110100 | 116 | t |
| 0110101 | 53 | 5 | 1010101 | 85 | U | 1110101 | 117 | u |
| 0110110 | 54 | 6 | 1010110 | 86 | V | 1110110 | 118 | V |
| 0110111 | 55 | 7 | 1010111 | 87 | W | 1110111 | 119 | w |
| 0111000 | 56 | 8 | 1011000 | 88 | $X$ | 1111000 | 120 | X |
| 0111001 | 57 | 9 | 1011001 | 89 | Y | 1111001 | 121 | y |
| 0111010 | 58 | : | 1011010 | 90 | Z | 1111010 | 122 | z |


| 0111011 | by | ; | 1011011 | Y1 | [ | 1111011 | 123 |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0111100 | 60 | < | 1011100 | 92 | 1 | 1111100 | 124 |  |
| 0111101 | 61 | $=$ | 1011101 | 93 | ] | 1111101 | 125 | \} |
| 0111110 | 62 | > | 1011110 | 94 | $\wedge$ | 1111110 | 126 | $\sim$ |
| 0111111 | 63 | ? | 1011111 | 95 | - |  |  |  |

In addition to visible characters like Z, \$, or 5, the encoding includes numbers for several special characters. Ex: A newline character is encoded as 10. Because no visible character exists for a newline, the language uses an escape sequence. An escape sequence is a two-character sequence starting with \that represents a special character. Ex: '\n' represents a newline character. Escape sequences also enable representing characters like ', ", or \. Ex: myChar = '\'' assigns myChar with a single-quote character. myChar = '<br>' assigns myChar with \ (just '\' would yield a compiler error, since $\backslash$ ' is the escape sequence for ', and then a closing ' is missing).

Table 2.10.2: Common escape sequences.

| Escape sequence | Char |
| :---: | :---: |
| $\backslash n$ | newline |
| $\backslash t$ | tab |
| $\backslash{ }^{\prime}$ | single quote |
| $\backslash "$ | double quote |
| $\backslash \backslash$ | backslash |


|  | Participation <br> Activity | 2.10.5: Character encoding. |
| :--- | :--- | :--- |
| \# | Question |  |
|  | The statement char keyPressed $=$ ' R ' stores <br> what decimal number in the memory location for <br> keyPressed? | $\square$ |

### 2.10.1: Printing a message with ints and chars.

Print a message telling a user to press the letterToQuit key numPresses times to quit. End with newlir numPresses $=2$, print:

Press the q key 2 times to quit.

```
import java.util.Scanner;
public class QuitScreen {
    public static void main (String [] args) {
        char letterToQuit = '?';
        int numPresses = 0;
        /* Your solution goes here */
        return;
    }
}
```

Run

## Section 2.11 - String basics

Some variables should store a sequence of characters like the name Julia. A sequence of characters is called a string. A string literal uses double quotes as in "Julia". Various characters may be included, such as letters, numbers, spaces, symbols like \$, etc., as in "Hello ... Julia!!".


Type a string below to see how a string is stored as a sequence of characters in memory (in this case, the string happens to be allocated to memory locations 501 to 506).


A programmer defines a string variable similarly to defining char, int, or double variables, but using the String data type. Note the capital S.

## Figure 2.11.1: Strings example: Word game.

```
import java.util.Scanner;
/* A game inspired by "Mad Libs" where user enters nouns,
    verbs, etc., and then a story using those words is output.
*/
public class StoryGame \{
    public static void main (String [] args) \{
            Scanner scnr = new Scanner(System.in);
            String wordRelative = "";
            String wordFood = "";
            String wordAdjective = "";
            String wordTimePeriod = " ";
            // Get user's words
            System.out.println("Provide input without spaces.");
            System.out.println("Enter a kind of relative: ");
            wordRelative = scnr.next();
            System.out.println("Enter a kind of food: ");
            wordFood = scnr.next();
            System.out.println("Enter an adjective: ");
            wordAdjective = scnr.next();
            System.out.println("Enter a time period: ");
            wordTimePeriod = scnr.next();
            // Tell the story
            System.out. println();
            System.out.print ("My " + wordRelative);
            System.out.println(" says eating " + wordFood);
            System.out.println("will make me more " + wordAdjective + ",");
            System.out.println("so now I eat it every " + wordTimePeriod + ".");
            return;
    \}
\}
```

Note that scnr.next() is used to get the next string from input, versus scnr.nextlnt() to get the next integer.

|  | Participation <br> Activity | 2.11.2: Strings. |
| :--- | :--- | :--- |
| \# | Question |  |
| 1 | Define a string named firstName. Don't initialize the <br> string. | $\square$ |
| 2 | Print a string named firstName, using println. | $\square$ |
| 3 | Read an input string from scnr into firstName. | $\square$ |

A programmer can initialize a string variable during definition, as in String firstMonth = "January";

scnr.next() gets the next input string only up to the next input space, tab, or newline. So following the user typing Betty Sue (ENTER), scnr.next) will only store Betty in stringVar. Sue will be the next input string. In contrast, the method scnr.nextLine() reads all user text on the input line, up to the newline character resulting from the user pressing ENTER, into stringVar.

## Figure 2.11.2: Reading an input string containing spaces using nextLine.

```
import java.util.Scanner;
public class NameWelcome {
    public static void main (String [] args) {
        Scanner scnr = new Scanner(System.in);
        String firstName = "";
        String lastName = "";
        System.out.println("Enter first name: ");
        firstName = scnr.nextLine(); // Gets enter line up to ENTER
        System.out.println("Enter last name: ");
        lastName = scnr.nextLine(); // Gets enter line up to ENTER
        System.out.println();
        System.out.println("Welcome " + firstName + " " + lastName + "!");
        System.out.println("May I call you " + firstName + "?");
        return;
    }
}
```

(An interesting poem about Sue McKay on YouTube (4 min)).

(ENTER) means the user presses the enter/return key. scnr is already defined.

| \# | Question | Your answer |
| :---: | :---: | :---: |
| 1 | Asked to enter a fruit name, the user types: <br> Fuji Apple (ENTER). <br> What does fruitName $=$ scnr. next() store in fruitName? |  |
| 2 | Given: ```System.out.println( "Enter fruit name:"); fruitName = scnr.next(); System.out.println( "Enter fruit color:"); fruitColor = scnr.next();``` <br> The user will type Fuji Apple (ENTER) for the fruit name and red (ENTER) for the fruit color. What is stored in fruitColor? |  |
| 3 | Using scnr, type a statement that reads an entire user-entered line of text into string userStr. |  |



## Participation Activity

2.11.5: Reading string input.

The following program is part of a larger application to get a user's mailing address. Run the program and observe the output. Update the program to store the entire mailing address in userAddress.

```
```

import java.util.Scanner;

```
```

import java.util.Scanner;
public class ReadMailingAddress {
public class ReadMailingAddress {
public static void main (String [] args) {
public static void main (String [] args) {
Scanner scnr = new Scanner(System.in);
Scanner scnr = new Scanner(System.in);
String userAddress = "";
String userAddress = "";
System.out.println("Enter street address: ");
System.out.println("Enter street address: ");
userAddress = scnr.next();
userAddress = scnr.next();
System.out.println("Street address is: " + userAdd
System.out.println("Street address is: " + userAdd
return;
return;
}
}
}

```
```

}

```
```

17

A String variable is a reference type (discussed in depth elsewhere) variable that refers to a String object. An object consists of some internal data items plus operations that can be performed on that data. Ex: String movieTitle = "Frozen"; defines a String reference variable named movieTitle that refers to a String object. That String object stores the string "Frozen".

A programmer can assign a new literal to a String variable, which creates a new String object. Ex: movieTitle = "The Martian" ; creates a new String object with the string "The Martian", and assigns the String object's reference to the variable movieTitle.

Assigning one String variable to another String variable causes both variables to refer to the same String, and does not create a new String. Ex: movieTitle = favoriteMovie; assigns favoriteMovie's reference to movieTitle. Both movieTitle and favoriteMovie refer to the same String object.

## Figure 2.11.3: Assigning a value to a string.

```
import java.util.Scanner;
public class SentenceFromStrings {
    public static void main(String[] args) {
            Scanner scnr = new Scanner(System.in);
            String userNoun1 = "";
            String userVerb = "";
            String userNoun2 = " ";
            String sentenceSubject = " ";
            String sentenceObject = "";
            System.out.print("Enter a noun: ");
            userNoun1 = scnr.next();
            System.out.print("Enter a verb: ");
            userVerb = scnr.next();
            System.out.print("Enter a noun: ");
            userNoun2 = scnr.next();
            sentenceSubject = userNoun1;
            sentenceObject = userNoun2;
            System.out.print(sentenceSubject);
            System.out.print(" ");
            System.out.print(userVerb);
            System.out.print(" ");
            System.out.print(sentenceObject);
            System.out.println(".");
            sentenceSubject = userNoun2;
            sentenceObject = userNoun1;
            System.out.print(sentenceSubject);
            System.out.print(" ");
            System.out.print(userVerb);
            System.out.print(" ");
            System.out.print(sentenceObject);
            System.out.println(".");
            return;
    }
}
```


## $-\square$ <br> Participation Activity <br> 2.11.6: Assigning a value to a String variable.

str1 and str2 are String variables.

| $\#$ | Question | Your answer |
| :--- | :--- | :--- |
| 1 | Write a statement that assigns "miles" to str1. | $\square$ |
| 2 | str1 is initially "Hello", str2 is "Hi". <br> After str1 = str2, what is str1? <br> Omit the quotes. | $\square$ |
| 3 | str1 is initially "Hello", str2 is "Hi". <br> After str1 = str2 and then str2 = "Bye", what is str1? <br> Omit the quotes. | $\square$ |

### 2.11.1: Reading and printing a string.

A user types a word and a number. Read them into the provided variables. Then print: word_number user entered: Amy 5

Amy_5

```
import java.util.Scanner;
public class SpaceReplace {
    public static void main (String [] args) {
        Scanner scnr = new Scanner(System.in);
        String userWord = "";
        int userNum = 0;
        /* Your solution goes here */
        return;
    }
}
```

Run

## Section 2.12 - Integer overflow

An integer variable cannot store a number larger than the maximum supported by the variable's data type. An overflow occurs when the value being assigned to a variable is greater than the maximum value the variable can store.

A common error is to try to store a value greater than about 2 billion into an int variable. For example, the decimal number 4,294,967,297 requires 33 bits in binary, namely 100000000000000000000000000000001 (we chose the decimal number for easy binary viewing).

Trying to assign that number into an int results in overflow. The 33rd bit is lost and only the lower 32 bits are stored, namely 00000000000000000000000000000001, which is decimal number 1 .


Defining the variable of type long, (described in another section) which uses 64 bits, would solve the above problem. But even that variable could overflow if assigned a large enough value.

Most compilers detect when a statement assigns a variable with a literal constant that is so large as to cause overflow. For example, the javac compiler reports the error "possible loss of precision".

A common source of overflow involves intermediate calculations. Given int variables num1, num2, num3 each with values near 1 billion, (num1 + num2 + num3) / 3 will encounter overflow in the numerator, which will reach about 3 billion (max int is around 2 billion), even though the final result after dividing by 3 would have been only 1 billion. Dividing earlier can sometimes solve the problem, as in (num1 / 3) + (num2 / 3) + (num3 / 3), but programmers should pay careful attention to possible implicit type conversions.


Run the program and observe the output is as expected. Replicate the multiplication and printing three more times, and observe incorrect output due to overflow. Change num's type to long, and observe the corrected output.

```
public class OverflowExample {
    public static void main (String [] args) {
        int num = 1000;
        num = num * 100;
        System.out.println("num: " + num);
        num = num * 100;
        System.out.println("num: " + num);
        num = num * 100;
        System.out.println("num: " + num);
        return;
    }
}
```

18


Assume all variables below are defined as int, which uses 32 bits.

| \# | Question | Your answer |
| :---: | :---: | :---: |
| 1 | Overflow can occur at any point in the program, and not only at a variable's initialization. | Yes |
|  |  | No |
| 2 | Will $x=1234567890$ cause overflow? | Yes |
|  |  | No |
| 3 | Will $\mathrm{x}=9999999999$ cause overflow? | Yes |
|  |  | No |
| 4 | Will $x=4000000000$ cause overflow? | Yes |
|  |  | No |
| 5 | Will these assignments cause overflow?$\begin{aligned} & x=1000 ; \\ & y=1000 ; \\ & z=x^{*} y ; \end{aligned}$ | Yes |
|  |  | No |
| 6 | Will these assignments cause overflow?$\begin{aligned} & x=1000 ; \\ & y=1000 ; \\ & z=x^{*} x ; \\ & z=z^{*} y^{*} y ; \end{aligned}$ | Yes |
|  |  | No |

## Section 2.13 - Numeric data types

int and double are the most common numeric data types. However, several other numeric types exist. The following table summarizes available integer numeric data types.

## Table 2.13.1: Integer numeric data types.

| Definition | Size | Supported number range |
| :--- | :--- | :--- |
| byte myVar; | 8 bits | -128 to 127 |
| short myVar; | 16 bits | $-32,768$ to 32,767 |
| int myVar; | 32 bits | $-2,147,483,648$ to $2,147,483,647$ |
| long myVar; | 64 bits | $-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$ |

int is the most commonly used integer type.
long is used for integers expected to exceed about 2 billion.
In case the reader is wondering, the language does not have a simple way to print numbers with commas. So if $x$ is 8000000 , printing $8,000,000$ is not trivial.

A common error made by a program's user is to enter the wrong type, such as entering a string when the input statement was myInt $=$ scnr. nextInt(); where mylnt is an int, which can cause strange program behavior.
short is rarely used. One situation is to save memory when storing many (e.g., tens of thousands) of smaller numbers, which might occur for arrays (another section). Another situation is in embedded computing systems having a tiny processor with little memory, as in a hearing aid or TV remote control. Similarly, byte is rarely used, except as noted for short.
$\left[\nabla \begin{array}{|c|c|}\substack{\text { Participation } \\ \text { Activity }} & \text { 2.13.1: Integer types. }\end{array}\right.$

Indicate whether each is a good variable definition for the stated purpose, assuming int is usually used for integers, and long is only used when absolutely necessary.

| \# | Question | Your answer |
| :--- | :--- | :--- |
| 1 | The number of days of school per year: |  |
| int numDaysSchoolyear; |  |  |
| 2 | The number of days in a human's lifetime. <br> int numDaysLife; | True |
| 3 | The number of years of the earth's existence. <br> int numYearsEarth; | True |
|  | False |  |
| The number of human heartbeats in one year, assuming 100 <br> beats/minute. <br> long numHeartBeats; | True |  |

The following table summarizes available floating-point numeric types.

## Table 2.13.2: Floating-point numeric data types.

| Definition | Size | Supported number range |
| :--- | :--- | :--- |
| float $x ;$ | 32 bits | $-3.4 \times 10^{38}$ to $3.4^{\star} 10^{38}$ |
| double $x ;$ | 64 bits | $-1.7 \times 10^{308}$ to $1.7^{\star} 10^{308}$ |

The compiler uses one bit for sign, some bits for the mantissa, and some for the exponent. Details are beyond our scope.
float is typically only used in memory-saving situations, as discussed above for short.
Due to the fixed sizes of the internal representations, the mantissa (e.g, the 6.02 in 6.02 e 23 ) is limited to about 7 significant digits for float and about 16 significant digits for double. So for a variable defined as double pi, the assignment pi $=3.14159265$ is OK, but pi $=3.14159265358979323846$ will be truncated.

A variable cannot store a value larger than the maximum supported by the variable's data type. An overflow occurs when the value being assigned to a variable is greater than the maximum value the variable can store. Overflow with floating-point results in infinity. Overflow with integer is discussed elsewhere.


On some processors, especially low-cost processors intended for "embedded" computing, like systems in an automobile or medical device, floating-point calculations may run slower than integer calculations, such as 100 times slower. Floating-point types are typically only used when really necessary. On more powerful processors like those in desktops, servers, smartphones, etc., special floating-point hardware nearly or entirely eliminates the speed difference.

Floating-point numbers are sometimes used when an integer exceeds the range of the largest integer type.

|  | Participation <br> Activity | 2.13.3: Floating-point numeric types. |  |
| :--- | :--- | :--- | :--- |
| $\#$ | Question | Your answer |  |
| 1 | float is the most commonly-used floating-point type. | True |  |
| 2 | int and double types are limited to about 16 digits. | False |  |
| 2 |  | False |  |

## Section 2.14 - Random numbers

Some programs need to use a random number. For example, a program might serve as an electronic dice roller, generating random numbers between 1 and 6 . The following example demonstrates how to generate four random numbers between 1 and 6 . The program's relevant parts are explained further below.

## Figure 2.14.1: Random numbers: Four dice rolls.

```
import java.util.Random;
public class DiceRoll { % Four roll:
    public static void main (String[] args) {
        Random randGen = new Random(); // New random number generator
        System.out.println("Four rolls of a dice...");
        // randGen.nextInt(6) yields 0, 1, 2, 3, 4, or 5
        // so + 1 makes that 1, 2, 3, 4, 5, or 6
        System.out.println(randGen.nextInt(6) + 1);
        System.out.println(randGen.nextInt(6) + 1);
        System.out.println(randGen.nextInt(6) + 1);
        System.out.println(randGen.nextInt(6) + 1);
        return;
    }
}
```

Line 1 makes Java's Random class available to the program. Line 5 creates a new random number generator object named randGen. The method call randGen.nextlnt(X) can then be used to get a random number ranging from 0 to $X-1$. Mentioned concepts like class, object, and method call will be described in later sections; here, the programmer can just copy the given code to get random numbers.

After the above setup, line 11 uses randGen.nextlnt(6) to get a new random number between 0 and 5 . The code adds 1 to obtain values between 1 and 6 . Lines 12, 13, and 14 follow similarly.


| 2 |  | 10 |
| :---: | :---: | :---: |
|  |  | Unknown |
| 3 | What is the largest possible value returned by randGen.nextInt(10)? | 10 |
|  |  | 9 |
|  |  | 11 |
| 4 | randGen.nextInt(), with no number between the (), returns a random value that could be any integer (of int type), positive or negative. What is the largest possible value? | 10 |
|  |  | 2,147,483,647 |
|  |  | Unknown |
| 5 | Which generates a random number in the range 18..30? | randGen.nextlnt(30) |
|  |  | randGen.nextlnt(31) |
|  |  | randGen.nextlnt(30 - 18) |
|  |  | randGen.nextlnt(30 - $\text { 18) }+18$ |
|  |  | $\begin{aligned} & \text { randGen.nextlnt(30- } \\ & 18+1)+18 \end{aligned}$ |

Because an important part of testing or debugging a program is being able to have the program run exactly the same across multiple runs, most programming languages use a pseudo-random number generation approach. A pseudo-random number generator produces a specific sequence of numbers based on a seed number, that sequence seeming random but always being the same for a
given seed. For example, a program that prints four random numbers and that seeds a random number generator with a seed of 3 might then print 99, 4, 55, and 7 . Running with a seed of 8 might yield $42,0,22,9$. Running again with 3 will yield $99,4,55$, and 7 again-guaranteed.

Early video games used a constant seed for "random" features, enabling players to breeze through a level by learning and then repeating the same winning moves.

Random() seeds the pseudo-random number generator with a number based on the current time; that number is essentially random, so the program will get a different pseudo-random number sequence on each run. On the other hand, Random(num) will seed the generator with the value num, where num is any integer (actually, any "long" value).

Having seen the current time's use as a random seed, you might wonder why a program can't just use a number based on the current time as a random number-why bother with a pseudo-random number generator at all? That's certainly possible, but then a program's run could never be identically reproduced. By using a pseudo-random number generator, a programmer can set the seed to a constant value during testing or debugging.


Type two statements using nextlnt() to print two random integers between 0 and 9 . End with a new 5

7

Note: For this activity, using one statement may yield different output (due to the compiler calling nex statements for this activity.

```
import java.util.Scanner;
import java.util.Random;
public class DiceRoll {
    public static void main (String [] args) {
        Random randGen = new Random();
        int seedVal = 0;
        randGen.setSeed(seedVal);
        /* Your solution goes here */
        return;
    }
}
```

Run

### 2.14.2: Fixed range of random numbers.

Type two statements that use nextlnt() to print 2 random integers between (and including) 100 and

## 112

102
Note: For this activity, using one statement may yield different output (due to the compiler calling nex statements for this activity.

```
import java.util.Scanner;
import java.util.Random;
public class RandomGenerateNumbers {
    public static void main (String [] args) {
        Random randGen = new Random();
        int seedVal = 0;
        seedVal = 4;
        randGen.setSeed(seedVal);
        /* Your solution goes here */
        return;
    }
}
```

Run

## Section 2.15 - Reading API documentation

Java provides an extensive set of classes for creating programs. Oracle's Java API Specification provides detailed documents describing how to use those classes. The class' documentation is known as an $\boldsymbol{A P I}$, short for application programming interface.

The main page of the Java documentation lists all Java packages that are available to programmers. A package is a group of related classes. Organizing classes into packages helps programmers find needed classes.

Figure 2.15.1: Java documentation lists and describes available packages.


Previous programs in this material used a Scanner object to read input from the user. The Scanner class is located in the package java.util. The Java documentation for a class consists of four main elements. The following uses the Scanner class to illustrate these documentation elements. The documentation for the Scanner is located at: Scanner class documentation.

Class overview: The first part of the documentation provides an overview of the class, describing the class' functionality and providing examples of how the class is commonly used in a program.

Figure 2.15.2: Scanner class' overview.

Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method
java.util
Class Scanner
java.lang.Object
java.util.Scanner
All Implemented Interfaces:
Closeable, AutoCloseable, Iterator<String>
public final class Scanner
extends Object
implements Iterator<String>, Closeable
A simple text scanner which can parse primitive types and strings using regular expressions.
A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

For example, this code allows a user to read a number from System. in:
Scanner sc = new Scanner(System.in); int i = sc.nextInt();

As another example, this code allows long types to be assigned from entries in a file myNumbers:

The package in which a class is located appears immediately above the class name. The figure above shows the Scanner class is located in the java.util package. To use a class, a program must include an import statement that informs the compiler of the class' location.

```
Construct 2.15.1: Import statement.
import packageName.ClassName;
```

The statement import java.util.Scanner; imports the scanner class.
Constructor summary: Provides a list and brief description of the constructors that can be used to create objects of the class.

## Figure 2.15.3: Scanner class' constructor summary.

## Constructor Summary

```
Constructors
Constructor and Description
Scanner(File source)
Constructs a new Scanner that produces values scanned from the specified file.
Scanner(File source, String charsetName)
Constructs a new Scanner that produces values scanned from the specified file.
Scanner(InputStream source)
Constructs a new Scanner that produces values scanned from the specified input stream.
Scanner(InputStream source, String charsetName)
Constructs a new Scanner that produces values scanned from the specified input stream.
Scanner(Path source)
Constructs a new Scanner that produces values scanned from the specified file.
Scanner(Path source, String charsetName)
Constructs a new Scanner that produces values scanned from the specified file.
Scanner(Readable source)
Constructs a new Scanner that produces values scanned from the specified source.
Scanner(ReadableByteChannel source)
Constructs a new Scanner that produces values scanned from the specified channel.
Scanner(ReadableByteChannel source, String charsetName)
Constructs a new Scanner that produces values scanned from the specified channel.
Scanner(String source)
Constructs a new Scanner that produces values scanned from the specified string.
```

Previous programs in this material used the statement
Scanner scnr = new Scanner(System.in); to construct a Scanner object. System.in is a InputStream object automatically created when a Java programs executes. So, the constructor Scanner (InputStream source) listed in the documentation is the matching constructor.

Method summary: Provides a list and brief description of all methods that can be called on objects of the class. The Java documentation only lists the public methods that a program may use.

## Figure 2.15.4: Scanner class' method summary.

| Method Summary |  |
| :---: | :---: |
| Methods |  |
| Modifier and Type | Method and Description |
| void | close() <br> Closes this scanner. |
| Pattern | delimiter() <br> Returns the Pattern this Scanner is currently using to match delimiters. |
| String | findInLine(Pattern pattern) <br> Attempts to find the next occurrence of the specified pattern ignoring delimiters. |
| String | findInLine(String pattern) <br> Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters. |
| String | findWithinHorizon(Pattern pattern, int horizon) Attempts to find the next occurrence of the specified pattern. |
| String | findWithinHorizon(String pattern, int horizon) <br> Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters. |
| boolean | hasNext () <br> Returns true if this scanner has another token in its input. |
| boolean | hasNext (Pattern pattern) <br> Returns true if the next complete token matches the specified pattern. |
| boolean | hasNext(String pattern) <br> Returns true if the next token matches the pattern constructed from the specified string. |
| boolean | hasNextBigDecimal() <br> Returns true if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method. |
| boolean | hasNextBigInteger() <br> Returns true if the next token in this scanner's input can be interpreted as a BigInteger in the default radix using the nextBigInteger () method. |

Constructor and method details: Lastly, the documentation for a class provides a detailed description of all constructors and methods for the class. For each method, the documentation provides the method declaration, a description of the method, a list of parameters (if any), a description of the method's return value, and a list of possible exceptions the method may throw (discussed elsewhere).

The following shows the method details for the nextInt() method.

Figure 2.15.5: Scanner class' nextInt method documentation.

## nextInt

```
public int nextInt()
```

Scans the next token of the input as an int.
An invocation of this method of the form nextInt () behaves in exactly the same way as the invocation nextInt (radix), where radix is the default radix of this scanner.

Returns:
the int scanned from the input
Throws:
InputMismatchException - if the next token does not match the Integer regular expression, or is out of range
NoSuchelementException - if input is exhausted
IllegalStateException - if this scanner is closed


Participation Activity

### 2.15.1: Java API Documentation.

Using Oracle's Java API Specification, match the class to the correct package.


| Drag and drop above item | java.util |
| :--- | :--- |


|  | java.awt |
| :--- | :--- |


|  | java.io |
| :--- | :--- |


|  | java.math |
| :--- | :--- |

## Reset

## Section 2.16 - Debugging

Debugging is the process of determining and fixing the cause of a problem in a computer program. Troubleshooting is another word for debugging. Far from being an occasional nuisance, debugging is a core programmer task, like diagnosing is a core medical doctor task. Skill in carrying out a methodical debugging process can improve a programmer's productivity.

Figure 2.16.1: A methodical debugging process.


- Predict a possible cause of the problem
- Conduct a test to validate that cause
- Repeat

A common error among new programmers is to try to debug without a methodical process, instead staring at the program, or making random changes to see if the output is improved.

Consider a program that, given a circle's circumference, computes the circle's area. Below, the output area is clearly too large. In particular, if circumference is 10, then radius is $10 / 2$ * PI_VAL, so about 1.6. The area is then PI_VAL * 1.6 * 1.6, or about 8, but the program outputs about 775.

## Figure 2.16.2: Circle area program: Problem detected.

```
    import java.util.Scanner;
    public class CircumferenceToArea {
        public static void main (String [] args) {
            Scanner scnr = new Scanner(System.in);
            double circleRadius = 0.0;
            double circleCircumference = 0.0;
            double circleArea = 0.0;
            final double PI_VAL = 3.14159265;
            System.out.print("Enter circumference: ");
            circleCircumference = scnr.nextDouble();
                                    Enter circumference: 10
                                    Circle area is: 775.156914350:
            circleRadius = circleCircumference / 2 * PI_VAL;
            circleArea = PI_VAL * circleRadius * circleRadius;
            System.out.println("Circle area is: " + circleArea);
            return;
        }
}
```

First, a programmer may predict that the problem is a bad output statement. This prediction can be tested by adding the statement area $=999$; The output statement is OK, and the predicted problem is invalidated. Note that a temporary statement commonly has a "FIXME" comment to remind the programmer to delete this statement.

## Figure 2.16.3: Circle area program: Predict problem is bad output.

```
import java.util.Scanner;
public class CircumferenceToArea {
        public static void main (String [] args) {
            Scanner scnr = new Scanner(System.in);
            double circleRadius = 0.0;
            double circleCircumference = 0.0;
            double circleArea = 0.0;
            final double PI_VAL = 3.14159265;
            System.out.print("Enter circumference: ");
            circleCircumference = scnr.nextDouble();
            circleRadius = circleCircumference / 2 * PI_VAL;
            circleArea = PI_VAL * circleRadius * circleRadius;
        circleArea = 999; // FIXME delete
        System.out.println("Circle area is: " + circleArea);
        return;
    }
}
```

                                    Enter circumference: 0
                                    Circle area is: 999.0
    Next, the programmer predicts the problem is a bad area computation. This prediction is tested by assigning the value 0.5 to radius and checking to see if the output is 0.7855 (which was computed by hand). The area computation is OK, and the predicted problem is invalidated. Note that a temporary statement is commonly left-aligned to make clear it is temporary.

Figure 2.16.4: Circle area program: Predict problem is bad area computation.

```
import java.util.Scanner;
public class CircumferenceToArea {
        public static void main(String[] args) {
            Scanner scnr = new Scanner(System.in);
            double circleRadius = 0.0;
            double circleCircumference = 0.0;
            double circleArea = 0.0;
            final double PI_VAL = 3.14159265;
            System.out.print("Enter circumference: ");
            circleCircumference = scnr.nextDouble();
                                    Enter circumference: 0
                                    Circle area is: 0.7853981625
            circleRadius = circleCircumference / 2 * PI_VAL;
        circleRadius = 0.5; // FIXME delete
            circleArea = PI_VAL * circleRadius * circleRadius;
            System.out.println("Circle area is: " + circleArea);
            return;
        }
}
```

The programmer then predicts the problem is a bad radius computation. This prediction is tested by assigning PI_VAL to the circumference, and checking to see if the radius is 0.5 . The radius computation fails, and the prediction is likely validated. Note that unused code was temporarily commented out.

Figure 2.16.5: Circle area program: Predict problem is bad radius computation.

```
import java.util.Scanner;
public class CircumferenceToArea {
        public static void main(String[] args) {
        Scanner scnr = new Scanner(System.in);
        double circleRadius = 0.0;
        double circleCircumference = 0.0;
        double circleArea = 0.0;
        final double PI_VAL = 3.14159265;
        System.out.print("Enter circumference: ");
        circleCircumference = scnr.nextDouble();
circleCircumference = PI_VAL; // FIXME delete
        circleRadius = circleCircumference / 2 * PI_VAL;
System.out.println("Radius: " + circleRadius); // FIXME delete
        /*
            area = PI_VAL * radius * radius;
            System.out.println("Circle area is: " + area);
            */
        return;
    }
}
```

The last test seems to validate that the problem is a bad radius computation. The programmer visually examines the expression for a circle's radius given the circumference, which looks fine at first glance. However, the programmer notices that radius = circumference / 2 * PI_VAL; should have been radius = circumference / ( 2 * PI_VAL); The parentheses around the product in the denominator are necessary and represent the desired order of operations. Changing to radius = circumference / (2 * PI_VAL); solves the problem.

The above example illustrates several common techniques used while testing to validate a predicted problem:

- Manually set a variable to a value.
- Insert print statements to observe variable values.
- Comment out unused code.
- Visually inspect the code (not every test requires modifying/running the code).

Statements inserted for debugging must be created and removed with care. A common error is to forget to remove a debug statement, such as a temporary statement that manually sets a variable to a value. Left-aligning such a statement and/or including a FIXME comment can help the programmer
remember. Another common error is to use /* */ to comment out code that itself contains /* */ characters. The first */ ends the comment before intended, which usually yields a syntax error when the second */ is reached or sooner.

The predicted problem is commonly vague, such as "Something is wrong with the input values." Conducting a general test (like printing all input values) may give the programmer new ideas as to a more-specific predicted problems. The process is highly iterative-new tests may lead to new predicted problems. A programmer typically has a few initial predictions, and tests the most likely ones first.


Use the above repeating two-step process (predict problem, test to validate) to find the problem in the following code.


| Answer based on the above discussion. | Participation <br> Activity | 2.16.2: Debugging. |  |
| :--- | :--- | :--- | :--- |
| \# | Question <br> The first step in debugging is to make random changes to the <br> code and see what happens. | True |  |
| 2 | A common predicted-problem testing approach is to insert <br> print statements. | True |  |
| 3 | Variables in temporary statements can be written in <br> uppercase, as in MYVAR $=999$, to remind the programmer to <br> remove them. | True |  |
| 4 | A programmer lists all possible predicted problems first, then <br> runs tests to validate each. | True |  |
| 5 | False |  |  |
| 2 | Most beginning programmers naturally follow a methodical <br> process. | True |  |
| A program's output should be positive and usually is, but in <br> some instances the output becomes negative. Overflow is a <br> good prediction of the problem. | True |  |  |

## Section 2.17 - Style guidelines

Each programming team, whether a company or a classroom, may have its own style for writing code, sometimes called a style guide. Below is the style guide followed by most code in this material. That style is not necessarily better than any other style. The key is to be consistent in style so that code within a team is easily understandable and maintainable.

You may not have learned all of the constructs discussed below; you may wish to revisit this section after covering new constructs.

## Table 2.17.1: Sample style guide.

| Sample guidelines, <br> used in this <br> material | Yes |  |
| :--- | :--- | :--- | :--- |
| Whitespace |  | No (for our sample style) |


| inconsistently if code is copied to different editors. (Auto-tabbing may need to be disabled in some source code editors). | $\} \quad \overline{\mathrm{y}}=\overline{\mathrm{x}}+1 ;$ | ```} if (a < b) { x = 25; // No }``` |
| :---: | :---: | :---: |
| Braces |  |  |
| For branches, loops, methods, or classes, opening brace appears at end of the item's line. Closing brace appears under item's start. | ```if (a<b) { // Called "K&R" style } while (x < y) { // K&R style }``` | ```if (a<b) { // Also popular, but we }``` |
| For if-else, the else appears on its own line | ```if (a < b) { } else { // "Stroustrup" style, }``` | ```if (a<b) { } else { // Original K&R style }``` |
| Braces always used even if only one substatement | $\begin{aligned} & \text { if }(a<b)\{ \\ & \quad x=25 ; \\ & \} \end{aligned}$ | $\text { if } \begin{aligned} &(a<b) \\ & x=25 ; \quad / / \text { No, can lei } \end{aligned}$ |
| Naming |  |  |
| Variable/parameter names are camelCase, starting with lowercase | int numItems; | ```int NumItems; // No int num_items; // Common``` |
| Variable/parameter names are descriptive, use at least two words (if possible, to reduce conflicts), and avoid abbreviations unless widelyknown like "num". Single-letter | int numboxes; char userKey; | ```int boxes; // No int b; // No char k; // No char usrKey; // No``` |


| varıadies are rare; exceptions for loop indices (i, j), or math items like point coordinates ( $\mathrm{x}, \mathrm{y}$ ). |  |  |
| :---: | :---: | :---: |
| Constants use upper case and underscores (and at least two words) | final int MAXIMUM_WEIGHT $=300$; | final int MAXIMUMWEIGHT = final int maximumWeight $=$ final int MAXIMUM $=300$; |
| Variables usually defined early (not within code), and initialized to be safe (if practical). | int i $=0$; <br> char userKey = '-'; | ```int i; // No char userKey; // No userKey = 'c'; int j; // No``` |
| Method names are camelCase with lowercase first. | printHello() | ```PrintHello() // No print_hello() // No``` |

## Miscellaneous

Lines of code are typically less than 100 characters wide.

Code is more easily readable when lines are kept short. One long line can usually be broken up into several smaller ones.
$\boldsymbol{K} \boldsymbol{R} \boldsymbol{R}$ style for braces and indents is named after C language creators Kernighan and Ritchie.
Stroustrup style for braces and indents is named after C++ language creator Bjarne Stroustrup. The above are merely example guidelines.

Exploring further:

- More on indent styles from Wikipedia.org
- Oracle's Java Style Guide


## Section 2.18 - Java example: Salary calculation with variables

Using variables in expressions, rather than numbers like 40, makes a program more general and
makes expressions more meaningful when read too.

## Participation

 Activity2.18.1: Calculate salary: Generalize a program with variables and input.

The following program uses a variable workHoursPerWeek rather than directly using 40 in the salary calculation expression.

1. Run the program, observe the output. Change 40 to 35 (France's work week), and run again.
2. Generalize the program further by using a variable workWeeksPerYear. Run the program. Change 50 to 52, and run again.
3. Introduce a variable monthlySalary, used similarly to annualSalary, to further improve program readability.
```
Reset
1
public class Salary {
3 public static void main (String [] args) {
4 int hourlyWage = 20;
5 int workHoursPerWeek = 40;
6 // FIXME: Define and initialize variable workWeeksPerYear, then replace the 5l
7 int annualSalary =0;
8
9 annualSalary = hourlyWage * workHoursPerWeek * 50;
10 System.out.print("Annual salary is: ");
11 System.out.println(annualSalary);
12
16 return;
17 }
18 }
19
```


## Run

When values are stored in variables as above, the program can read user inputs for those values. If a value will never change, the variable can be defined as final.


Participation Activity
2.18.2: Calculate salary: Generalize a program with variables and input.

The program below has been generalized to read a user's input value for hourlyWage.

1. Run the program. Notice the user's input value of 10 is used. Modify that input value, and run again.
2. Generalize the program to get user input values for workHoursPerWeek and workWeeksPerYear (change those variables' initializations to 0). Run the program.
3. monthsPerYear will never change, so define that variable as final. Use the standard for naming final variables. Ex: final int MAX_LENGTH = 99. Run the program.
4. Change the values in the input area below the program, and run the program again.

## Reset

1
2 import java.util.Scanner;
3
4 public class Salary \{ public static void main (String [] args) \{

Scanner scnr = new Scanner(System.in);
int hourlyWage $=0$;
int workHoursPerWeek = 40;
int workWeeksPerYear = 50;
int monthsPerYear $=12$; // FIXME: Define as final and use standard naming
int annualSalary $=0$;
int monthlySalary $=0$;



```
                hourlyWage = scnr.nextInt();
                // FIXME: Get user input values for workHoursPerWeek and workWeeksPerYear
            17
            18
            19
                annualSalarv = hourlvWaae * workHoursPerWeek * workWeeksPerYear:
10
```

Run

## Section 2.19 - Java example: Married-couple names with variables



Pat Smith and Kelly Jones are engaged. What are possible last name combinations for the married couple (listing Pat first)?

1. Run the program below to see three possible married-couple names. Note the use of variable firstNames to hold both first names of the couple.
2. Extend the program to define and use a variable lastName similarly. Note that the print statements are neater. Run the program again.
3. Extend the program to print two more options that abut the last names, as in SmithJones and JonesSmith. Run the program again.

## Reset

1 imnort iava.util.Scanner:

```
2
public class ShowMarriedNames {
4
5 public static void main(String[] args) {
6 Scanner scnr = new Scanner(System.in);
    String firstName1 = "";
    String lastName1 = "";
    String firstName2 = "";
    String lastName2 = "";
    String firstNames = "";
    // FIXME: Define lastName
    System.out.println("What is the first person's first name?");
    firstName1 = scnr.nextLine();
    System.out.println("What is the first person's last name?");
    lastName1 = scnr.nextLine();
    Svstem.out.orintln("What is the second person's first name?"):
```

Pat
Smith
Kelly

Run


A solution to the above problem follows:

## Reset

import java.util.Scanner;
2
3 public class ShowMarriedNames \{
4
5 public static void main(String[] args) \{
6 Scanner scnr = new Scanner(System.in);
$7 \quad$ String firstName1 = "";
8 String lastName1 = "";
$9 \quad$ String firstName2 = "";
10 String lastName2 = "";
11 String firstNames = "";
12 String lastName = "";
13
14 System.out.println("What is the first person's first name?");
15 firstName1 = scnr.nextLine();
16 System.out.println("What is the first person's last name?");
17 lastName1 = scnr.nextLine();
18
19 Svstem.out.println("What is the second person's first name?"):
Pat
Smith
Kelly

