#### Chapter 2 - Variables / Assignments

## Section 2.1 - Variables (int)

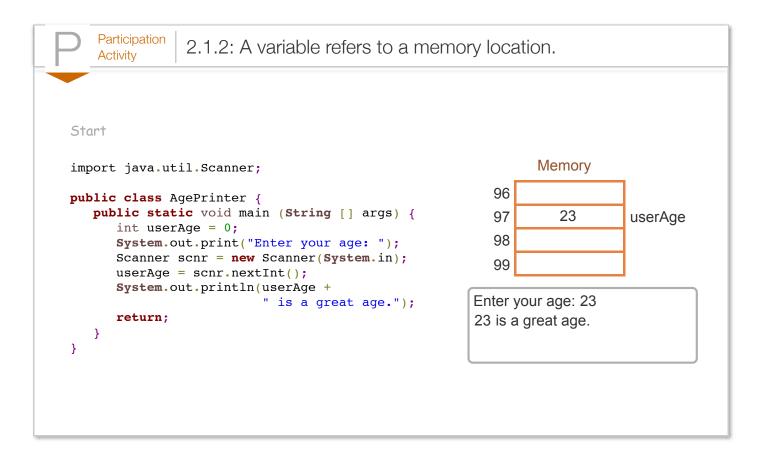
Here's a variation on a common schoolchild riddle.

Participati Activity	2.1.1: People on bus.
For each step, l editable).	keep track of the current number of people by typing in the numPeople box (it's
	Start
	You are driving a bus. The bus starts with 5 people.
	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.")

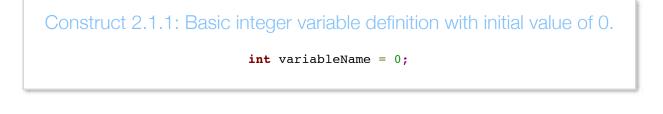
A *variable* represents a memory location used to store data. That location is like the "box" that you used above. The statement *int userAge; defines* (also called *declares*) a new variable named userAge. The compiler allocates a memory location for userAge capable of storing an integer, hence the "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.<sup>mem</sup> 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.

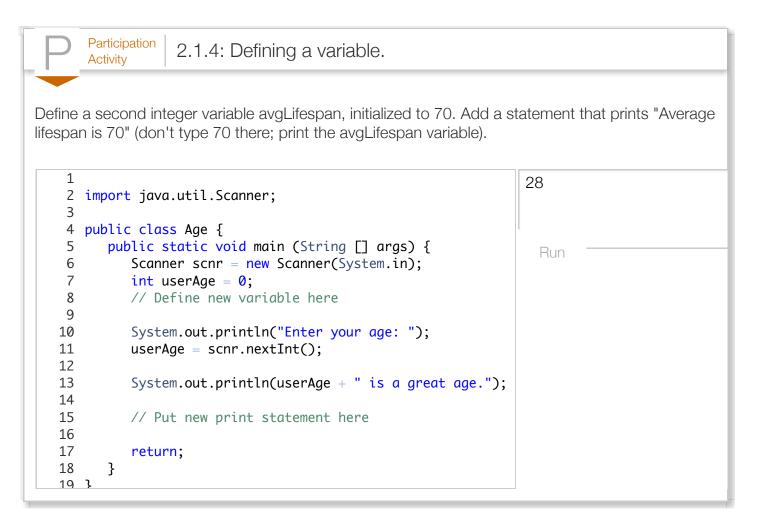
Although not required, an integer variable is commonly assigned an initial value when defined.



F	Participation Activity2.1.3: Defining integer variables	
Note	e: Capitalization matters, so MyNumber is not the sam	e as myNumber.
#	Question	Your answer
1	Define an integer variable named numPeople. Do not initialize the variable.	
2	Define an integer variable named numDogs, initializing the variable to 0 in the definition.	
3	Define an integer variable named daysCount, initializing the variable to 365 in the definition.	
4	What memory location (address) will a compiler allocate for the variable definition: int numHouses = 99; If appropriate, type: Unknown	

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.

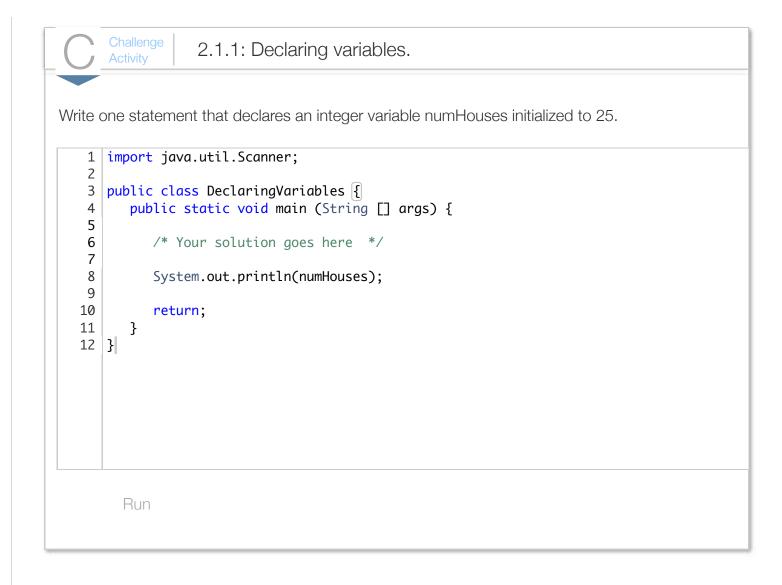


A <u>common error</u> is to read a variable that has not yet been assigned a value. If a local variable is defined but not initialized, the variable's memory location contains some unknown value, commonly but not always 0. A program with an uninitialized variable may thus run correctly on system that 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 <u>good practice</u> 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.

	Ρ	Participation Activity 2.1.5: int variables.	
V	Vhic	n statement is an error?	
	#	Question	Your answer
	1	int dogCount;	Error
	Ι		No error
	0	int amountOwed = -999;	Error
	2		No error
	0	int numYears = 9000111000;	Error
	3		No 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).



(\*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).<sup>asgn</sup>

Construct 2.2.1: Assignment statement.

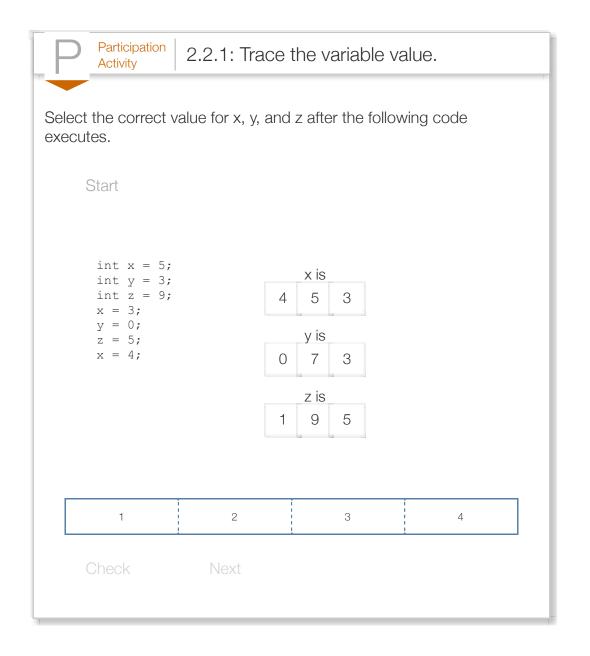
variableName = expression;

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,");
                                                             One female mouse may give
                                                             and up to 140 mice, in a y
        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.





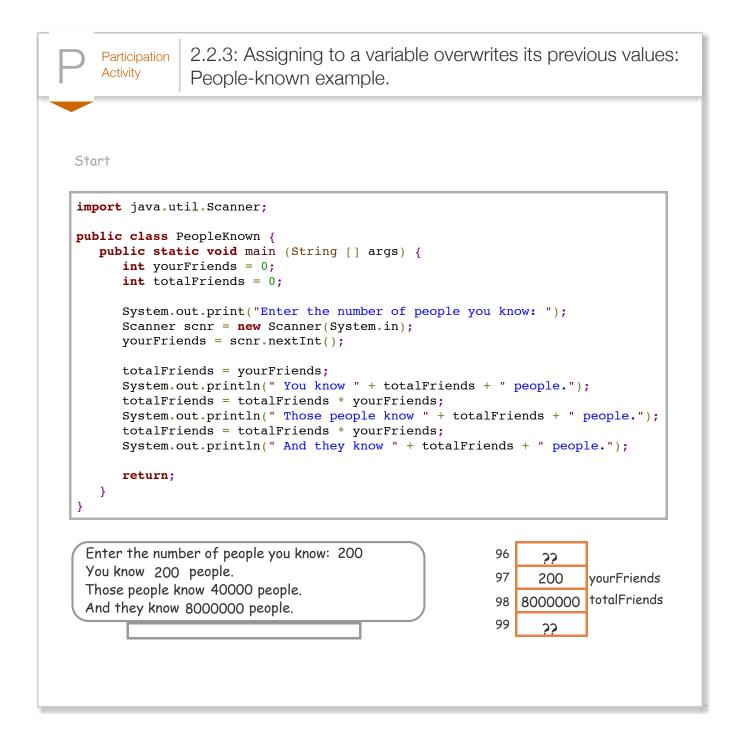
Ρ	Participation Activity 2.2.2: Assignment statements.	
Be si	ure to end assignment statements with a semicolon ;.	
#	Question	Your answer
1	Write an assignment statement to assign 99 to numCars.	

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

A <u>common error</u> 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 "numItems EQUALS numApples", but this material strives to avoid such inaccurate wording.

Another <u>common error</u> 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.

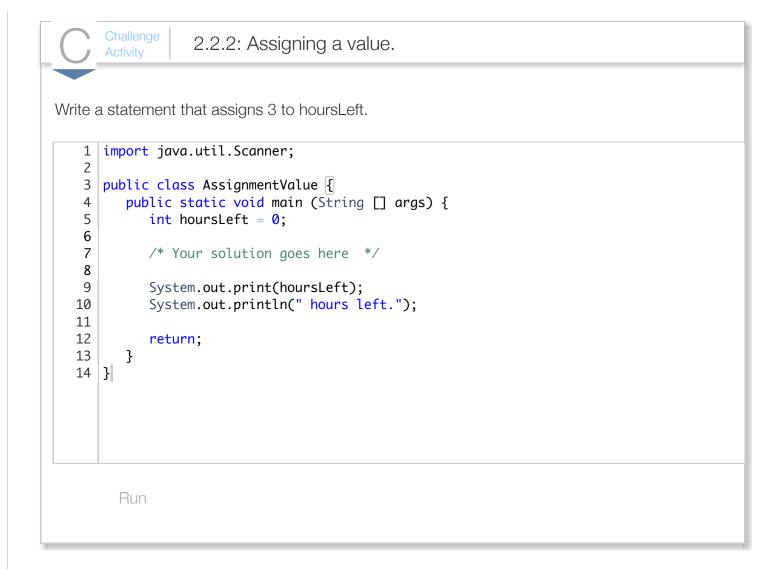


(The above example relates to the popular idea that any two people on earth are connected by just "six degrees of separation", accounting for overlapping of known-people.

	Question	Your answer
1	<pre>numApples is initially 5. What is numApples after: numApples = numApples + 3;</pre>	
2	<pre>numApples is initially 5. What is numFruit after: numFruit = numApples; numFruit = numFruit + 1;</pre>	
3	Write a statement ending with - 1 that decreases variable flyCount's value by 1.	

#	Question	Your answer
11	w = 1; y = 2; z = 4;	
1	x = y + 1; w = 2 - x; z = w * y;	
	x = 4; y = 0; z = 3;	
2	x = x - 3; y = y + x; z = z * y;	
	x = 6; y = -2;	
3	y = x + x; w = y * x; z = w - y;	
	w = -2; x = -7; y = -8;	
4	z = x - y; z = z * w; z = z / w;	

Challen Activity		er the outpu	ut of the v	variable as	ssignmer	nts.
Start						
Otart						
	Enter the	output of th	e followin	g prograr	n.	
public s int x int y x = 9	= 6; ; n.out.print(x + "	String [] ar		96		
1	2	3	4	5		6
Check	Next					



Challenge 2.2.3: Assigning a sum.

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

```
2
 3
   public class AssigningSum {
 4
      public static void main (String [] args) {
 5
      int numCoins = 0;
 6
      int numNickels = 0;
 7
      int numDimes = 0;
8
9
      numNickels = 5;
10
      numDimes = 6;
11
      /* Your solution goes here */
12
13
      System.out.print("There are ");
14
15
      System.out.print(numCoins);
16
      System.out.println(" coins");
17
18
      return;
19
      }
20 }
```

Run

Challenge 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 1 import java.util.Scanner; 2 3 public class AssigningNumberToVariable { public static void main (String [] args) { 4 5 int numPeople = 0; 6 7 numPeople = 10;8 /\* Your solution goes here \*/ 9 10 System.out.print("There are "); 11 12 System.out.print(numPeople); 13 System.out.println(" people."); 14 15 return; 16 } 17 } 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 <u>good practice</u> followed by many Java programmers is to not use \_ or \$ in programmer-created 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.	
n are valid identifiers?	
Question	Your answer
numCars	Valid
	Invalid
num_Cars1	Valid
	Invalid
_numCars	Valid
	Invalid
numCars	Valid
	Invalid
num cars	Valid
	Invalid
	n are valid identifiers? Question numCars num_Cars1numCarsnumCars

	3rdPlace	Valid
6		Invalid
7	thirdPlace_	Valid
/		Invalid
8	thirdPlace!	Valid
0		Invalid
9	tall	Valid
9		Invalid
10	short	Valid
		Invalid
11	very tall	Valid
		Invalid

Ρ	Participation Activity2.3.2: Identifier validator.	
Note: I	Doesn't consider library items.	
Try an	identifier:	Validate
Awaitir	ng 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 <u>good practice</u> 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 numItemsOnShelves. The following are less meaningful: age (whose age?), sqft (what's that stand for?), num (almost no info). <u>Good practice</u> 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 <u>good practice</u> 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.

ŧ	Question	Your answer
	The number of students attending UCLA.	num
		numStdsUcla
1		numStudentsUcla
		numberOfStudentsAttendingUc
	The size of an LCD monitor	size
		sizeLcdMonitor
2		S
		sizeLcdMtr
	The number of jelly beans in a jar.	numberOfJellyBeansInTheJar
		JellyBeansInJar
3		jellyBeansInJar

assertfinallypublicbooleanfloatreturnbreakforshortbytegotostaticcaseifstrictfpcatchimplementssupercharinstanceofsynchronizedconstinterfacethiscontinueinterfacethrowsdefaultlongtransientdoublenewtryelsepackagevoid	abstract	final	protected
breakforshortbytegotostaticbytegotostaticcaseifstrictfpcatchimplementssupercharimportswitchclassinstanceofsynchronizedconstinterfacethiscontinueinterfacethrowsdefaultlongthrowsdoublenewtryelsepackagevoid	assert	finally	public
bytegotostaticcaseifstrictfpcatchimplementssupercharimportswitchclassinstanceofsynchronizedconstinterfacethiscontinueinterfacethrowsdefaultlongthrowsdoublenewtryelsepackagevoid	boolean	float	return
caseifstrictfpcatchimplementssupercharimportswitchclassinstanceofsynchronizedconstintthiscontinueinterfacethrowdefaultlongthrowsdonativetransientdoublenewtryelsepackagevoid	break	for	short
catchimplementssupercharimportswitchclassinstanceofsynchronizedconstintthiscontinueinterfacethrowdefaultlongthrowsdonativetransientdoublenewtryelsepackagevoid	byte	goto	static
charimportswitchclassinstanceofsynchronizedconstintthiscontinueinterfacethrowdefaultlongthrowsdonativetransientdoublenewtryelsepackagevoid	case	if	strictfp
classinstanceofsynchronizedconstintthiscontinueinterfacethrowdefaultlongthrowsdonativetransientdoublenewtryelsepackagevoid	catch	implements	super
constintthiscontinueinterfacethrowdefaultlongthrowsdonativetransientdoublenewtryelsepackagevoid	char	import	switch
continueinterfacethrowdefaultlongthrowsdonativetransientdoublenewtryelsepackagevoid	class	instanceof	synchronized
defaultlongthrowsdonativetransientdoublenewtryelsepackagevoid	const	int	this
donativetransientdoublenewtryelsepackagevoid	continue	interface	throw
doublenewtryelsepackagevoid	default	long	throws
else package void	do	native	transient
	double	new	try
enum private volatile	else	package	void
	enum	private	volatile
extends	extends		

# 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).

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.

	Question	Your answer
	Is the following an expression? 12	Yes
		No
	ls the following an expression? int eggsInCarton	Yes
2		No
3	ls the following an expression? eggsInCarton * 3	Yes
		No
	Is the following an error? An int's maximum value is 2,147,483,647.	Yes
-	numYears = 1,999,999,999;	No

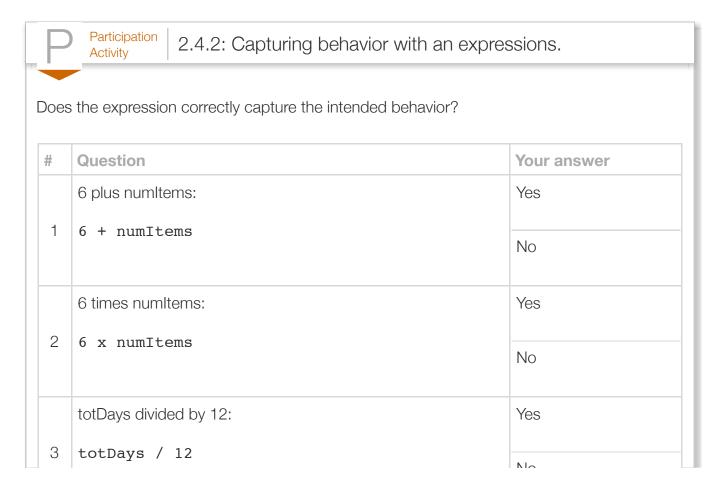
An **operator** is a symbol for a built-in language calculation like + for addition. **Arithmetic operators** built into the language are:

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

Modulo may be unfamiliar and is discussed further below.

Parentheses may be used, as in: ((userItems + 1) \* 2) / totalItems. 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 5y to represent 5 times y.



		INU
	5 times i:	Yes
4	5i	No
	The negative of userVal:	Yes
5	-userVal	No
	itemsA + itemsB, divided by 2:	Yes
6	itemsA + itemsB / 2	No
	n factorial	Yes
7	n!	No

```
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
                                                                        Enter down payr
                                                                        500
                                                                        Enter monthly ;
        System.out.println("Enter down payment: ");
                                                                        300
        downpayment = scnr.nextInt();
                                                                        Enter number o:
                                                                        60
        System.out.println("Enter monthly payment: ");
                                                                        Total cost: 18!
        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 <u>good practice</u> is to include a single space around operators for readability, as in numltems + 2, rather than numltems+2. An exception is - used as negative, as in: xCoord = -yCoord. - used as negative is known as **unary minus**.

ŀ		Participation Activity	2.4.3: Single space around oper	ators.		
Re	Retype each statement to follow the good practice of a single space around operators.					
#		Question		Your answer		
1		housesCity	= housesBlock *10;			
2	<u>)</u>	x = x1+x2+	2;			
Э	3	numBalls=n	umBalls+1;			
4	ļ	numEntries	= (userVal+1)*2;			

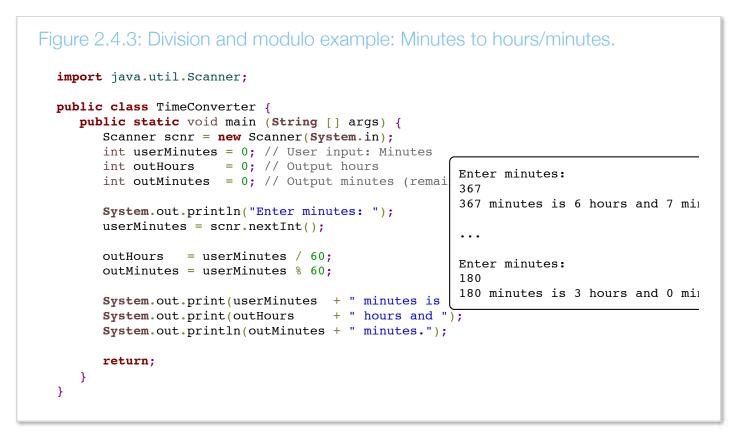
When the / operands are integers, the division operator / performs integer division, throwing away any remainder. Examples:

- 24 / 10 is 2.
- 50 / 50 is 1.
- 1 / 2 is 0. 2 divides into 1 zero times; remainder of 1 is thrown away.

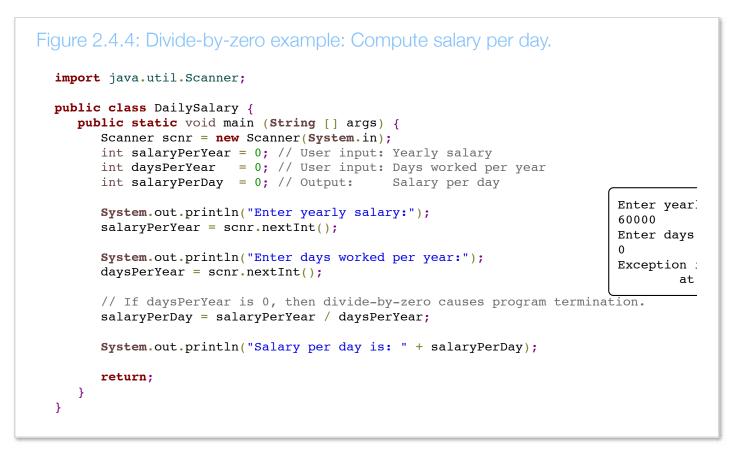
A <u>common error</u> is to forget that a fraction like (1 / 2) in an expression performs integer division, so the expression evaluates to 0.

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.



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.

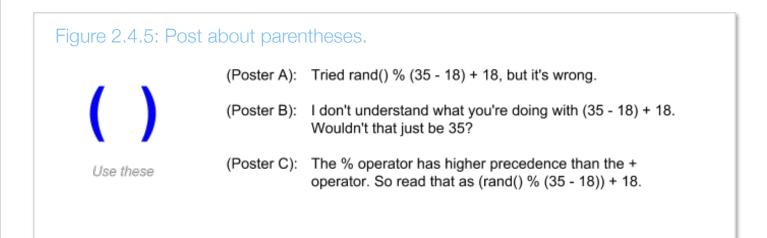


F	Participation Activity 2.4.4: Integer division and modulo.					
litera	ermine the result. Type "Error" if the program would terr als appear in these expressions to focus attention on th ide variables.					
#	Question	Your answer				
1	13 / 3					
2	4 / 9					
3	(5 + 10 + 15) * (1 / 3)					
4	50 % 2					
5	51 % 2					
6	78 % 10					
7	596 % 10					
8	100 / (1 / 2)					

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

Convention	Description	Explanation
()	Items within parentheses are evaluated first	In $2 \times (A + 1)$ , A + 1 is computed first, with the result then multiplied by 2.
unary -	- used as a negative (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-to- right	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 <u>common error</u> 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). <u>Good practice</u> 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.





2.4.5: Precedence rules.

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

#	Question	Your answer
	y + 2 * z	(y + 2) * z
1		y + (2 * z)
	z / 2-x	(z / 2) - x
2		z / (2 - x)
3	X * Y * Z	(x * y) * z
		x * (y * z)
4	x + y % 3	(x + y) % 3
		x + (y % 3)
5	x + 1 * y / 2	((x + 1) * y) / 2
		x + ((1 * y) / 2)
		x+ ( 1 * (y / 2))
	x / 2 + y / 2	((x / 2) + y) / 2
6		(x / 2) + (y / 2)

What is totCount after executing the follow numItems = 5;	ving? 44
7 totCount = 1 + (2 * numItems) * 4;	41

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 %=.

P	Participation Activity         2.4.6: Compound operators	
ap	propriate, type: Not possible	
ŧ	Question	Your answer
1	numAtoms is initially 7. What is numAtoms after: numAtoms += 5?	
2	numAtoms is initially 7. What is numAtoms after: numAtoms *= 2?	
3	Rewrite the statement using a compound operat carCount = carCount / 2;	or:
4	Rewrite the statement using a compound operat numItems = boxCount + 1;	or:

C Challenge Activity							
Start							
Enter the output of the following program.							
<pre>public class combinedOutput {     public static void main (String [] args) {         int x = 2;         int y = 0;         y = 3 * (x + 8);         System.out.print(x + " " + y);         return;     } }</pre>							
1 2 3 4 5							
Check	Next						

Challenge

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.

```
import java.util.Scanner;
 1
 2
 3
   public class ComputingFinalResult {
 4
      public static void main (String [] args) {
 5
         int num1 = 0;
 6
         int num2 = 0;
 7
         int finalResult = 0;
 8
9
         num1 = 4;
10
         num2 = 5;
11
         /* Your solution goes here */
12
13
         System.out.print("Final result: ");
14
         System.out.println(finalResult);
15
16
17
         return;
18
      }
19 }
```

Run

Challenge Activity

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.

```
public class ComputingChange {
 3
      public static void main (String [] args) {
 4
 5
         int amountToChange = 0;
 6
         int numFives = 0;
 7
         int numOnes = 0;
 8
9
         amountToChange = 19;
10
         numFives = amountToChange / 5;
11
         /* Your solution goes here */
12
13
14
         System.out.print("numFives: ");
         System.out.println(numFives);
15
         System.out.print("numOnes: ");
16
         System.out.println(numOnes);
17
18
19
         return;
      }
20
21 }
```

Run

Challenge

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.

```
import java.util.Scanner;
 1
 2
 3
   public class ComputingTotalCost {
      public static void main (String [] args) {
 4
 5
         int numDrinks = 0;
 6
         int numTacos = 0;
 7
         int totalCost = 0;
 8
9
         numDrinks = 4;
10
         numTacos = 6;
11
12
         /* Your solution goes here */
13
         System.out.print("Total cost: ");
14
         System.out.println(totalCost);
15
16
17
         return;
18
      }
19 }
```

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. <u>Good practice</u> 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);
                                                                     Enter a distance :
        double milesTravel = 0.0; // User input of miles to travel
                                                                     1800
        double hoursFly = 0.0; // Travel hours if flying those m
                                                                     1800.0 miles would
        double hoursDrive = 0.0; // Travel hours if driving those
                                                                     3.6 hours to fly,
                                                                     30.0 hours to driv
        System.out.print("Enter a distance in miles:\n");
        milesTravel = scnr.nextDouble();
                                                                     • • •
        hoursFly = milesTravel / 500.0;
                                                                     Enter a distance :
        hoursDrive = milesTravel / 60.0;
                                                                     400.5
                                                                     400.5 miles would
        System.out.println(milesTravel + " miles would take:");
                                                                     0.801 hours to fly
        System.out.println(hoursFly + " hours to fly,");
                                                                     6.675 hours to dr:
        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.

F	Participation Activity2.5.1: Defining and assigning	double variables.
All va	ariables are of type double and already-defined unle	ss otherwise noted.
#	Question	Your answer
1	Define a double variable named personHeight and initialize to 0.0.	
2	Compute ballHeight divided by 2.0 and assign the result to ballRadius. Do not use the fraction 1.0 / 2.0; instead, divide ballHeight directly by 2.0.	
3	Multiply ballHeight by the fraction one half, namely (1.0 / 2.0), and assign the result to ballRadius. Use the parentheses around the fraction.	

P	Participation Activity 2.5.2: Floating-point literals.	
#	Question	Your answer
	Which statement best defines and initializes the double variable?	double currHumidity = 99%;
1		double currHumidity = 99.0;
		double currHumidity = 99;
0	Which statement best assigns to the variable? Both variables are of type double.	cityRainfall = measuredRain - 5;
2		cityRainfall = measuredRain - 5.0;
	Which statement best assigns to the variable? cityRainfall is of type double.	cityRainfall = .97;
3		cityRainfall = 0.97;
· · · · · ·		I

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 6.02e23 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 1.0e-3. For a floating-point literal, <u>good practice</u> is to make the leading digit non-zero.

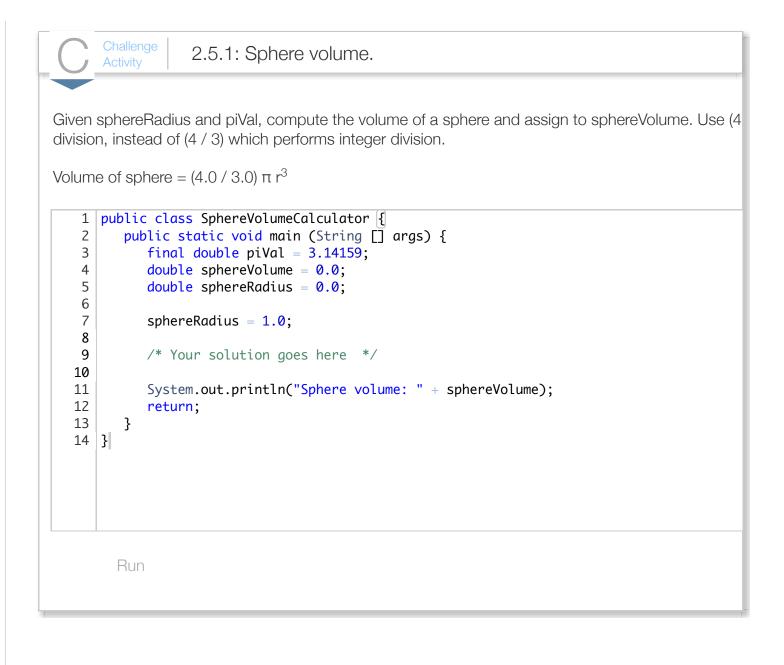
F	Participation Activity 2.5.3: Scientific notation.	
#	Question	Your answer
1	Type 1.0e-4 as a floating-point literal but not using scientific notation, with a single digit before and four digits after the decimal point.	
2	Type 7.2e-4 as a floating-point literal but not using scientific notation, with a single digit before and five digits after the decimal point.	
3	Type 540,000,000 as a floating-point literal using scientific notation with a single digit before and after the decimal point.	
4	Type 0.000001 as a floating-point literal using scientific notation with a single digit before and after the decimal point.	
5	Type 623.596 as a floating-point literal using scientific notation with a single digit before and five digits after the decimal point.	

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).

P	Participation Activity 2.5.4: Floating-point versus integer.	
Choo	ose the right type for a variable to represent each item.	
#	Question	Your answer
-	The number of cars in a parking lot.	double
1		int
	The current temperature in Celsius.	double
2		int
	A person's height in centimeters.	double
3		int
	The number of hairs on a person's head.	double
4		int
	The average number of kids per household.	double
5		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.

F	Participation Activity 2.5.5: Floating-point division.	
Dete	ermine the result.	
#	Question	Your answer
	13.0 / 3.0	4
1		4.333333
		Positive infinity
	0.0 / 5.0	0.0
2		Positive infinity
		Negative infinity
	12.0 / 0.0	12.0
3		Positive infinity
		Negative infinity
		Negative infinity





2.5.2: Acceleration of gravity.

Compute the acceleration of gravity for a given distance from the earth's center, distCenter, assigning expression for the acceleration of gravity is:  $(G * M) / (d^2)$ , where G is the gravitational constant 6.673 5.98 x 10<sup>24</sup> (in kg) and d is the distance in meters from the earth's center (stored in variable distCenter)

```
public class GravityCalculation {
 1
 2
      public static void main (String [] args) {
 3
         final double G
                            = 6.673e - 11;
 4
         final double M = 5.98e24;
 5
         double accelGravity = 0.0;
 6
         double distCenter = 0.0;
 7
 8
         distCenter = 6.38e6;
9
         /* Your solution goes here */
10
11
         System.out.println("accelGravity: " + accelGravity);
12
13
         return;
14
      }
15 }
     Run
```

# Section 2.6 - Constant variables

A <u>good practice</u> 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 <u>good practice</u>, 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.	
<pre>import java.util.Scanner;</pre>	
<pre>// Estimates distance of lightning based on seconds // between lightning and thunder</pre>	
<pre>public class LightningDist {     public static void main (String[] args) {         Scanner scnr = new Scanner(System.in);         final double SPEED_OF_SOUND = 761.207; // Miles/hour (set         final double SECONDS_PER_HOUR = 3600.0; // Secs/hour         double secondsBetween = 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;         System.out.println("Lightning strike was approximately");     }; }</pre>	ea level) Enter sec lightning 7 Lightning 1.4801247:  Enter sec lightning 1 Lightning 0.2114463
<pre>System.out.println(distInMiles + " miles away."); return; }</pre>	



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	<pre>int STEP_SIZE = 5;</pre>	True
		False
2	<pre>final int STEP_SIZE = 14;</pre>	True
		False
	<pre>totalStepHeight = numSteps * STEP_SIZE;</pre>	True
3		False
4	<pre>STEP_SIZE = STEP_SIZE + 1;</pre>	True
4		False

Challenge

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.

```
1 import java.util.Scanner;
 2
 3
   public class ShippingCalculator {
      public static void main (String [] args) {
 4
 5
         int shipWeightPounds = 10;
         int shipCostCents = 0;
 6
 7
         final int FLAT_FEE_CENTS = 75;
 8
         /* Your solution goes here */
9
10
11
         System.out.print("Weight(lb): " + shipWeightPounds);
         System.out.print(", Flat fee(cents): " + FLAT_FEE_CENTS);
12
         System.out.print(", Cents per pound: " + CENTS_PER_POUND);
13
         System.out.println(", Shipping cost(cents): " + shipCostCents);
14
15
16
         return;
17
      }
18 }
     Run
```

## Section 2.7 - Using math methods

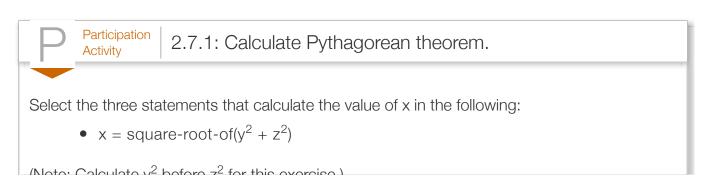
Some programs require math operations beyond basic operations like + and \*, such as computing a square root or raising a number to a power. Thus, Java comes with a standard *Math class* that has about 30 math operations available, listed later in this section. As shown below, the programmer first imports 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<sup>e</sup>.

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



```
https://zybooks.zyante.com/#/zybook/LehmanCMP167Spring2016/chapter/2/print
```

(ואטנב. טמוטעומנב א מבוטוב צ וטו נוווס באבוטוסב.)

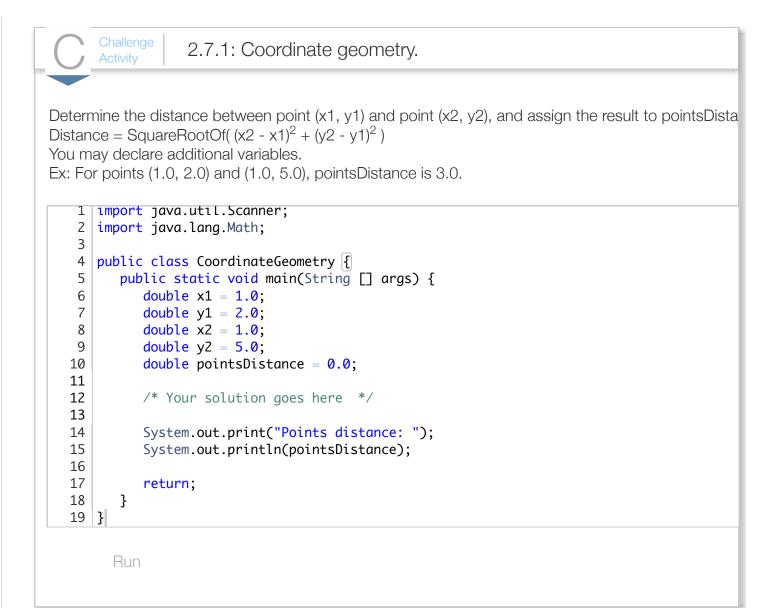
#	Question	Your answer
	First statement is:	temp1 = Math.pow(x , 2.0);
		temp1 = Math.pow(z , 3.0);
1		temp1 = Math.pow(y , 2.0);
		temp1 = Math.sqrt(y);
	Second statement is:	temp2 = Math.sqrt(x , 2.0);
		temp2 = Math.pow(z , 2.0);
2		temp2 = Math.pow(z);
		temp2 = x + Math.sqrt(temp1 + temp2);
	Third statement is:	temp2 = Math.sqrt(temp1 + temp2);
3		x = Math.pow(temp1 + temp2, 2.0);
		x = Math.sqrt(temp1) + temp2;
		x = Math.sqrt(temp1 + temp2);

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
log10	Common logarithm	atan	Arc tangent
log1p	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 floating- point division
		nextAfter	Next larger (or smaller) floating-point value
random	Generates random value between 0.0 and 1.0	nextUp	Next larger floating- point 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.

	ermine the final value of z for the following code se wer in the form 4.0.	gments. All variables are of type double.
ŧ	Question	Your answer
1	<pre>y = 2.3; z = 3.5; z = Math.ceil(y);</pre>	
2	<pre>y = 2.3; z = 3.5; z = Math.floor(z);</pre>	
3	<pre>y = 3.7; z = 4.5; z = Math.pow(Math.floor(z), 2.0);</pre>	
4	<pre>z = 15.75; z = Math.sqrt(Math.ceil(z));</pre>	
5	z = Math.abs(-1.8);	



Challenge

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, co

```
import java.Lang.Math;
 Ζ
 3
   public class TreeHeight {
 4
      public static void main(String [] args) {
 5
 6
         double treeHeight
                                = 0.0;
 7
         double shadowLength = 0.0;
 8
         double angleElevation = 0.0;
9
         angleElevation = 0.11693706; // 0.11693706 radians = 6.7 degrees
10
11
         shadowLength
                        = 17.5;
12
13
         /* Your solution goes here */
14
15
         System.out.print("Tree height: ");
16
         System.out.println(treeHeight);
17
18
         return;
19
      }
20 }
     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	2.8.1: Implicit conversions among double and int.	
0	on, given int numItems = 5, and double itemWeight = 0.5. For er to tenths, e.g., 8.0, 6.5, or 0.1.	
Question	Your answer	
3.0 / 1.5		
3.0 / 2		
(numltems + 10) / 2		
(numltems + 10) / 2.0		
	Activity 2.0.1. Implicit the value of the given expression loating-point answer, give answ Question 3.0 / 1.5 3.0 / 2 (numltems + 10) / 2	

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 per
        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 myIntVar 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).

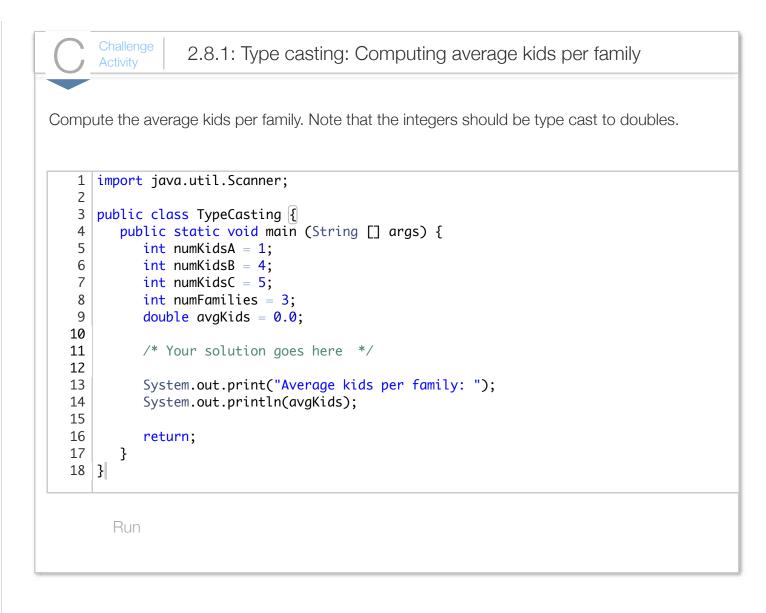
Such explicit conversion by the programmer of one type to another is known as *type casting*.

Figure 2.8.2: Using type casting to obtain floating-point division.	
<pre>import java.util.Scanner;</pre>	
<pre>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</pre>	
<pre>double avgKidsPerFamily = 0.0; // Expect fraction, so double avgKidsPerFamily = (double)(kidsInFamily1 + kidsInFamily2)</pre>	Average kids p
<pre>System.out.println("Average kids per family: " + avgKidsPerFam return; } </pre>	ily);

A <u>common error</u> 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: myInt = (int)myDouble. The fractional part is truncated. Ex: 9.5 becomes 9.

Participation Activity 2.8.2: Type casting.					
# Question	Your answer				
Which yields 2.5?	(int)(10) / (int)(4)				
1	(double)(10) / (double) (4)				
	(double)(10 / 4)				



### 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*.

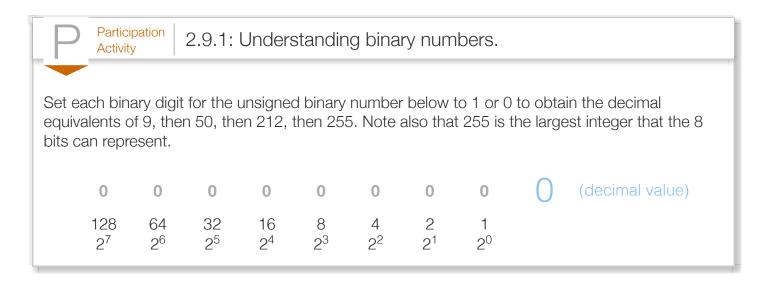
For a number in the more familiar base 10, known as a *decimal number*, each digit must be 0-9 and each digit's place is weighed by increasing powers of 10.

Decimal number with 3 digits	Representation
212	$2*10^{2} + 1*10^{1} + 2*10^{0} =$ 2*100 + 1*10 + 2*1 = 200 + 10 + 2 = 212

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

Binary number with 4 bits	Representation
1101	$1*2^{3} + 1*2^{2} + 0*2^{1} + 1*2^{0} =$ 1*8 + 1*4 + 0*2 + 1*1 = 8 + 4 + 0 + 1 = 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.



#	Question	Your answer
1	Convert the binary number 00001111 to a decimal number.	
2	Convert the binary number 10001000 to a decimal number.	
3	Convert the decimal number 17 to an 8-bit binary number.	
4	Convert the decimal number 51 to an 8-bit binary number.	

#### Section 2.10 - Characters

A variable of **char** type can store a single character, like the letter m or 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 + arrowHead >
            --->
            ooo>
        system.out.println("" + arrowBody + arrowBody + arrowHead);
        system.out.println("" + arrowBody + arrowBody + arrowHead);
        system.out.println("" + 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 <u>common error</u> is to use double quotes rather than single quotes around a character literal, as in myChar = "x", yielding a compiler error. Similarly, a <u>common error</u> is to forget the quotes around a character literal, as in myChar = x, usually yielding a compiler error.

F	Participation Activity 2.10.1: char data type.						
#	Question	Your answer					
1	In one statement, define a variable named userKey of type char and initialize to the letter a.						

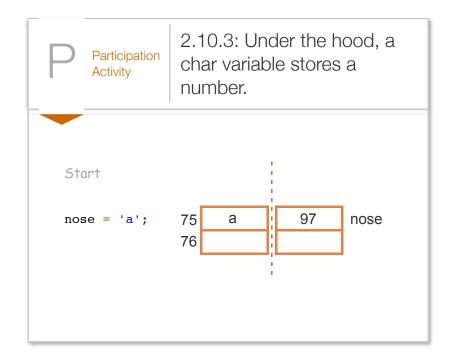


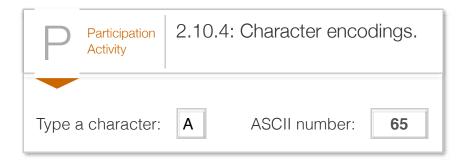
2.10.2: char variables.

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.

1		*
2	<pre>public class CharWarn{</pre>	
3	<pre>public static void main (String [] args) {</pre>	
4	<pre>char sepSym = '-';</pre>	1
5		Run
6	<pre>System.out.print("!WARNING!");</pre>	
7	<pre>System.out.print(" " + sepSym + sepSym + " ");</pre>	
8	<pre>System.out.print("!WARNING!");</pre>	
9	<pre>System.out.println("");</pre>	
10		
11	return;	
12	}	
13	}	
14		

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 0s on the left to yield 16 bits.

#### Table 2.10.1: Unaracter encodings as numbers in the ASUII standard.

Binary	Dec	Char	Binary	Dec	Char	Binary	Dec	Char
010 0000	32	space	100 0000	64	@	110 0000	96	`
010 0001	33	!	100 0001	65	А	110 0001	97	а
010 0010	34	н	100 0010	66	В	110 0010	98	b
010 0011	35	#	100 0011	67	С	110 0011	99	С
010 0100	36	\$	100 0100	68	D	110 0100	100	d
010 0101	37	%	100 0101	69	E	110 0101	101	е
010 0110	38	&	100 0110	70	F	110 0110	102	f
010 0111	39	I	100 0111	71	G	110 0111	103	g
010 1000	40	(	100 1000	72	Н	110 1000	104	h
010 1001	41	)	100 1001	73	Ι	110 1001	105	i
010 1010	42	*	100 1010	74	J	110 1010	106	j
010 1011	43	+	100 1011	75	K	110 1011	107	k
010 1100	44	,	100 1100	76	L	110 1100	108	I
010 1101	45	-	100 1101	77	М	110 1101	109	m
010 1110	46		100 1110	78	Ν	110 1110	110	n
010 1111	47	/	100 1111	79	0	110 1111	111	0
011 0000	48	0	101 0000	80	Р	111 0000	112	р
011 0001	49	1	101 0001	81	Q	111 0001	113	q
011 0010	50	2	101 0010	82	R	111 0010	114	r
011 0011	51	3	101 0011	83	S	111 0011	115	S
011 0100	52	4	101 0100	84	Т	111 0100	116	t
011 0101	53	5	101 0101	85	U	111 0101	117	u
011 0110	54	6	101 0110	86	V	111 0110	118	V
011 0111	55	7	101 0111	87	W	111 0111	119	W
011 1000	56	8	101 1000	88	Х	111 1000	120	х
011 1001	57	9	101 1001	89	Y	111 1001	121	У
011 1010	58	:	101 1010	90	Z	111 1010	122	Z

011 1011	59	;	101 1011	91	l	111 1011	123	{
011 1100	60	<	101 1100	92	$\setminus$	111 1100	124	
011 1101	61	=	101 1101	93	]	111 1101	125	}
011 1110	62	>	101 1110	94	$\wedge$	111 1110	126	~
011 1111	63	?	101 1111	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 = '\' assigns myChar with \ (just '\' would yield a compiler error, since \' is the escape sequence for ', and then a closing ' is missing).

Table 2.10.2: Common escape sequ					
Escape se	equence	Char			
\r	)	newline			
\t		tab			
\'		single quote			
\"		double quote			
		backslash			

F	Participation Activity 2.10.5: Character encoding.	
#	Question	Your answer
1	The statement <b>char keyPressed = 'R'</b> stores what decimal number in the memory location for keyPressed?	



#### 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!!".

Participation Activity	2.11.1: A string is stored as a sequer memory.	nce of characters in
	v to see how a string is stored as a sequence ppens to be allocated to memory locations 50	
	Type a string (up to 6 characters): Julia	Memory 501 J 502 U 503 I 504 i 505 a 506 J

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 = "";
                                                                         Provide input
        // Get user's words
                                                                         Enter a kind (
        System.out.println("Provide input without spaces.");
                                                                         mother
                                                                         Enter a kind (
        System.out.println("Enter a kind of relative: ");
                                                                         apples
        wordRelative = scnr.next();
                                                                         Enter an adjed
                                                                         loud
        System.out.println("Enter a kind of food: ");
                                                                         Enter a time 1
                                                                         week
        wordFood = scnr.next();
        System.out.println("Enter an adjective: ");
                                                                         My mother say:
        wordAdjective = scnr.next();
                                                                         will make me r
                                                                         so now I eat :
        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;
     }
  }
```

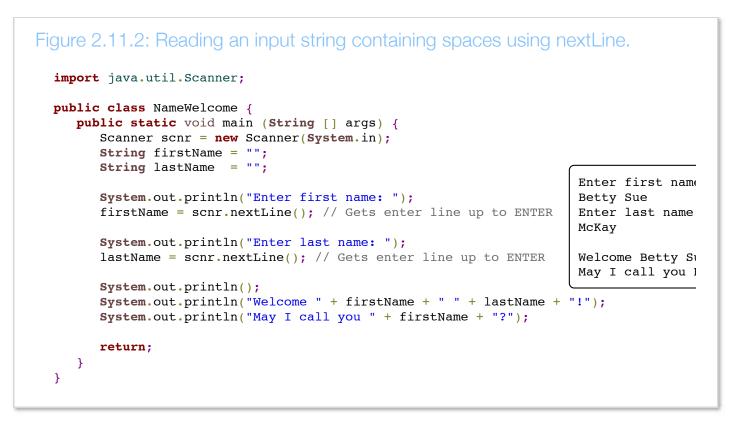
Note that scnr.next() is used to get the next string from input, versus scnr.nextInt() to get the next integer.

F	Participation Activity 2.11.2: Strings.	
#	Question	Your answer
1	Define a string named firstName. Don't initialize the string.	
2	Print a string named firstName, using println.	
3	Read an input string from scnr into firstName.	

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

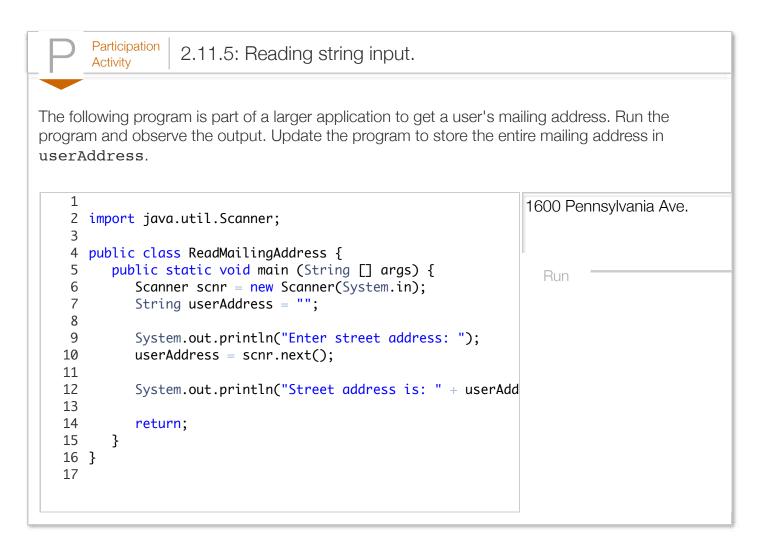
F	Participation Activity 2.11.3: String initialization.				
#	Question	Vour anowar			
#	Question	Your answer			
1	Define a string named smallestPlanet, initialized to "Mercury", using the above syntax.				

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.



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

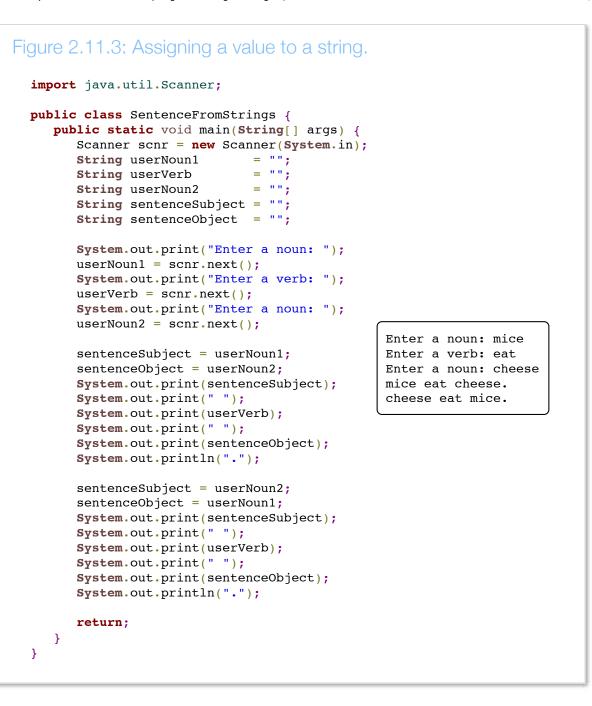
	TER) means the user presses the enter/return key. scnr	
ŧ	Question	Your answer
	Asked to enter a fruit name, the user types:	
	Fuji Apple (ENTER).	
1	What does fruitName = scnr.next() store in fruitName?	
	Given:	
2	<pre>System.out.println( "Enter fruit name:"); fruitName = scnr.next(); System.out.println( "Enter fruit color:"); fruitColor = scnr.next();</pre>	
-	The user will type <i>Fuji Apple (ENTER)</i> for the fruit name and <i>red (ENTER</i> ) 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.	



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.

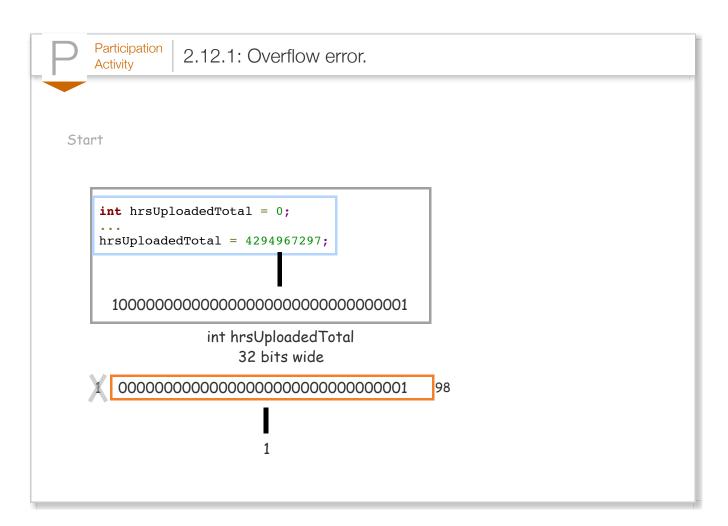


tr1	and str2 are String variables.	
#	Question	Your answer
1	Write a statement that assigns "miles" to str1.	
2	str1 is initially "Hello", str2 is "Hi". After str1 = str2, what is str1? Omit the quotes.	
3	str1 is initially "Hello", str2 is "Hi". After str1 = str2 and then str2 = "Bye", what is str1? Omit the quotes.	



## 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.



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.



2.12.2: long long variables.

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.

Run

```
1
 2 public class OverflowExample {
 3
      public static void main (String [] args) {
 4
         int num = 1000;
 5
 6
         num = num * 100;
 7
         System.out.println("num: " + num);
 8
9
         num = num * 100;
         System.out.println("num: " + num);
10
11
12
         num = num * 100;
13
         System.out.println("num: " + num);
14
15
         return;
16
      }
17 }
18
```

SSL	ime all variables below are defined as int, which uses 32 bits.	
ł	Question	Your answer
_	Overflow can occur at any point in the program, and not only at a variable's initialization.	Yes
1		No
	Will x = 1234567890 cause overflow?	Yes
2		No
	Will x = 9999999999 cause overflow?	Yes
3		No
	Will x = 400000000 cause overflow?	Yes
4		No
	Will these assignments cause overflow? x = 1000;	Yes
5	y = 1000; z = x * y;	No
	Will these assignments cause overflow? x = 1000;	Yes
6	x = 1000; y = 1000; z = x * x;	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.

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 <u>common error</u> 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 myInt 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.

_ [	$\supset$	Participation Activity2.13.1: Integer types.	
		e whether each is a good variable definition for the stated p or integers, and long is only used when absolutely necessar	
	#	Question	Your answer
	-	The number of days of school per year: int numDaysSchoolYear;	True
	1		False
	2	The number of days in a human's lifetime. int numDaysLife;	True
			False
		The number of years of the earth's existence. int numYearsEarth;	True
	3		False
		The number of human heartbeats in one year, assuming 100 beats/minute. long numHeartBeats;	100 True
	4		False

The following table summarizes available floating-point numeric types.

ble 2.13.2: Floating-point numeric data typ			
Definition	Size	Supported number range	
float x;	32 bits	-3.4x10 <sup>38</sup> to 3.4*10 <sup>38</sup>	
double x;	64 bits	-1.7x10 <sup>308</sup> to 1.7*10 <sup>308</sup>	

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 6.02e23) 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.

Participation Activity	2.13.2: Representation	on of floating-point (double) va	ilues.
Enter a decimal va	lue:		
Sign	Exponent         0       0       0       0       0	1.00000000	Mantissa 0 0 0 0 0 (

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.

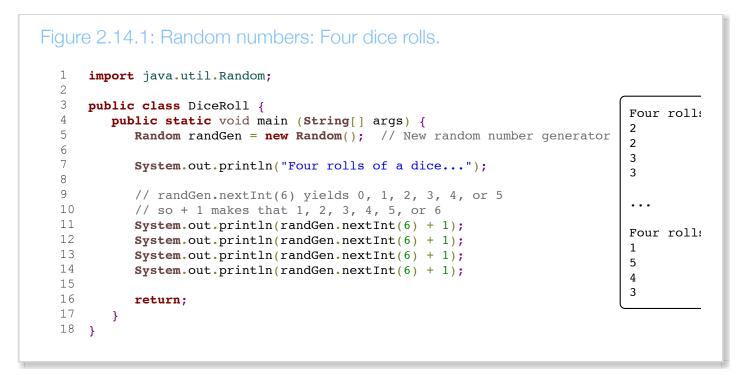
Floating-point numbers are sometimes used when an integer exceeds the range of the largest integer type.

Ρ	Participation Activity 2.13.3: Floating-point numeric types.	
#	Question	Your answer
4	float is the most commonly-used floating-point type.	True
I		False
0	int and double types are limited to about 16 digits.	True
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.





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.nextInt(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.nextInt(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.

P	Participation Activity 2.14.1: Random numbers.	
rand	Gen already exists.	
#	Question	Your answer
	If program is executing and randGen.nextInt(10) returns the value 6, what will the <i>next</i> randGen.nextInt(10) return?	7
I		Unknown
	What is the smallest <i>possible</i> value returned by randGen.nextInt(10)?	0
		1

-	2		10 Unknown
		What is the largest <i>possible</i> value returned by randGen.nextInt(10)?	10
	3		9
			11
		randGen.nextInt(), with no number between the (), returns a random value that could be <i>any</i> integer (of int type), positive or negative. What is the largest possible value?	10
	4		2,147,483,647
			Unknown
		Which generates a random number in the range 1830?	randGen.nextInt(30)
			randGen.nextInt(31)
	5		randGen.nextInt(30 - 18)
			randGen.nextInt(30 - 18) + 18
			randGen.nextInt(30 - 18 + 1) + 18

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.

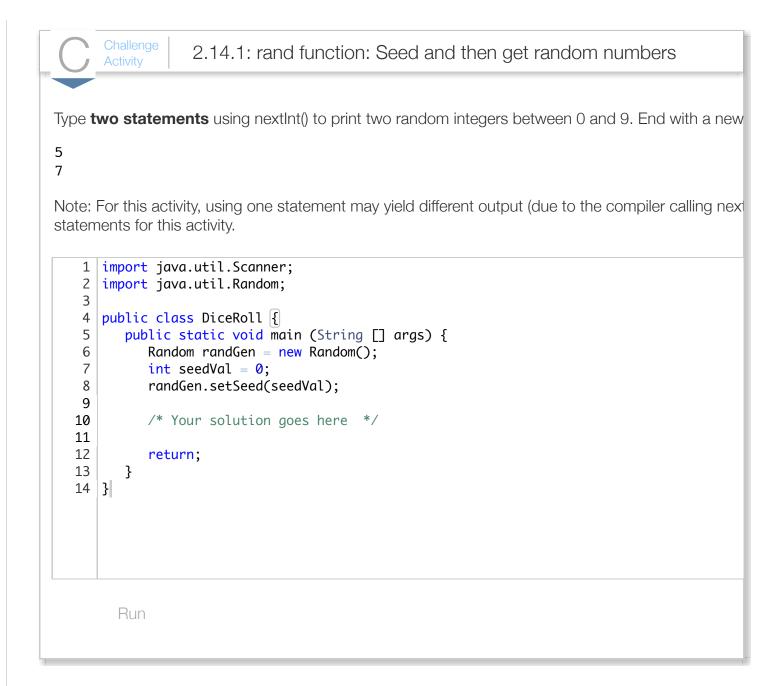
Early video games used a constant seed for "random" features, enabling players to breeze through a level by learning and then repeating the same winning moves.

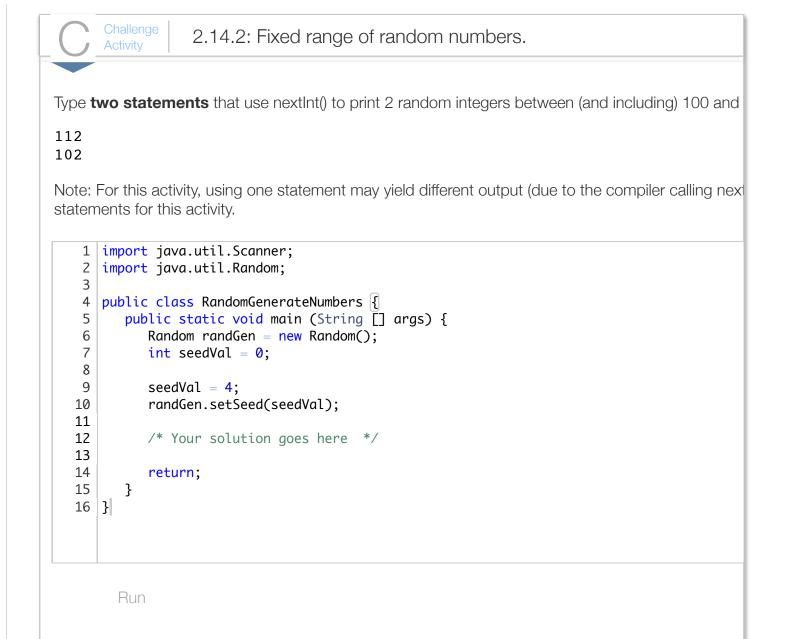


Random() seeds the pseudo-random number generator with a number based on the current time; that number is essentially random, so the program will get a different pseudo-random number sequence on each run. On the other hand, Random(num) will seed the generator with the value num, where num is any integer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

P	Participation Activity2.14.2: Seeding a pseudo-rando	om number generator.
#	Question	Your answer
1	A dice-rolling program has a statement that seeds a pseudo-random number generator with the constant value 99. The program is run and prints 4, 3, 6, 0. An hour later, the program is run again. What is the first number printed? Type a number or "Unknown" if the solution is unknown.	
2	A dice-rolling program's pseudo-random number generator is seeded with a number based on the current time. The program is run and prints 3, 2, 1, 6. An hour later, the program is run again. What is the first number printed? Type a number or "Unknown" if the solution is unknown.	





## 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 **API**, 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.

Java™ Platform Standard Ed. 7	Overview Package Class Use Tr	ee Deprecated Index Help Java™ Platform Standard Ed. 7	
All Classes	Prev Next Frames No Frames		
Packages	Java <sup>™</sup> Platform, Standard Edition 7 API Specification This document is the API specification for the Java <sup>™</sup> Platform, Standard Edition.		
ava.applet ava.awt ava.awt.color			
ava.awt.datatransfer			
ava.awt.event	See: Description		
All Classes			
	Packages		
AbstractAction AbstractAnnotationValueVisitor6	Package	Description	
AbstractAnnotationValueVisitor7 AbstractBorder AbstractButton	java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.	
AbstractCellEditor AbstractCollection	java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.	
AbstractColorChooserPanel	java.awt.color	Provides classes for color spaces.	
AbstractDocument AbstractDocument.AttributeContext	java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.	
AbstractDocument.Content AbstractDocument.ElementEdit AbstractElementVisitor6 AbstractElementVisitor7	java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.	
AbstractExecutorService AbstractInterruptibleChannel	java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.	
AbstractLayoutCache AbstractLayoutCache.NodeDimensi	java.awt.font	Provides classes and interface relating to fonts.	
AbstractList AbstractListModel AbstractMap	java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.	
AbstractMap.SimpleEntry	java.awt.im	Provides classes and interfaces for the input method framework.	
AbstractMap.SimpleImmutableEntry AbstractMarshallerImpl AbstractMethodError	java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.	
AbstractOwnableSynchronizer	java.awt.image	Provides classes for creating and modifying images.	
AbstractOwnableSynchronizer AbstractPreferences AbstractProcessor	java.awt.image java.awt.image.renderable	Provides classes for creating and modifying images. Provides classes and interfaces for producing rendering-independent images.	

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.

	use Tree Deprecated Index Help	Java™ Platform Standard Ed. 7
Prev Class Next Class	Frames No Frames All Classes	
Summary: Nested   Field   Co	onstr   Method Detail: Field   Constr   Method	
java.util		
<b>Class Scanner</b>		
java.lang.Object java.util.Scanner		
All Implemented Inter	faces:	
Closeable, AutoClos	eable, Iterator <string></string>	
public final clas extends Object	cor <string>, Closeable</string>	
Imprementes reerat		
-	nich can parse primitive types and strings using regular expressions.	
A simple text scanner wh	nich can parse primitive types and strings using regular expressions. put into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of diffe	rent types using the
A simple text scanner wh A scanner breaks its in various next methods.		rent types using the

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.

onstructo	or Summary	
Constructors		
	nd Description	
Scanner(Fil	,	
	ew Scanner that produces values scanned from the specified file.	
	e source, <b>String</b> charsetName) ew Scanner that produces values scanned from the specified file.	
	utStream source) ew Scanner that produces values scanned from the specified input stream.	
	utStream source, String charsetName) ew Scanner that produces values scanned from the specified input stream.	
Scanner (Pat	h source)	
Constructs a n	ew Scanner that produces values scanned from the specified file.	
	h source, <b>String</b> charsetName) ew Scanner that produces values scanned from the specified file.	
Scanner ( Rea	dable source)	
Constructs a n	ew Scanner that produces values scanned from the specified source.	
Scanner ( Rea	dableByteChannel source)	
Constructs a n	ew Scanner that produces values scanned from the specified channel.	
Scanner ( Rea	dableByteChannel source, String charsetName)	
Constructs a n	ew Scanner that produces values scanned from the specified channel.	
Scanner (Str	ing source)	
Constructs a n	ew 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.

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

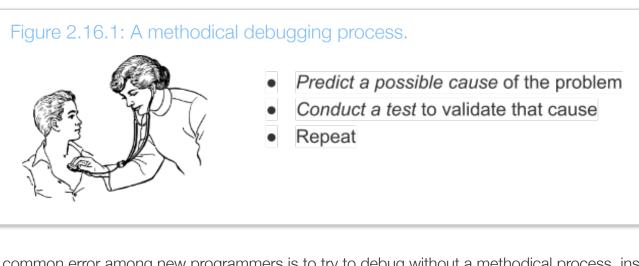
The following shows the method details for the nextInt() method.

14

nextInt	
<pre>public int nextInt()</pre>	
Scans the next token of the input as an int	t.
An invocation of this method of the form ne radix of this scanner.	extInt() behaves in exactly the same way as the invocation nextInt(radix), where radix is the default
Returns:	
the int scanned from the input	
Throws:	
	ext token does not match the Integer regular expression, or is out of range
NoSuchElementException - if input	
IllegalStateException - if this sca	anner is ciosed
Participation 0151.	
Activity 2.15.1:	Java API Documentation.
g Oracle's Java API Spec	ification, match the class to the correct package.
g Oracle's Java API Spec	ification, match the class to the correct package.
	Sification, match the class to the correct package.
BigDecimal FileInput	Stream Scanner Rectangle
	Stream Scanner Rectangle
BigDecimal FileInput	Stream Scanner Rectangle
BigDecimal FileInput	Stream Scanner Rectangle java.util
BigDecimal FileInput	Stream Scanner Rectangle
BigDecimal FileInput	Stream Scanner Rectangle java.util
BigDecimal FileInput	Stream Scanner Rectangle java.util java.awt
BigDecimal FileInput	Stream Scanner Rectangle java.util
BigDecimal FileInput	Stream Scanner Rectangle java.util java.awt
BigDecimal FileInput	Stream Scanner Rectangle   java.util java.awt java.io
BigDecimal FileInput	Stream Scanner Rectangle java.util java.awt
BigDecimal FileInput	Stream Scanner Rectangle   java.util java.awt java.io
BigDecimal FileInput	Stream Scanner Rectangle   java.util java.awt java.io

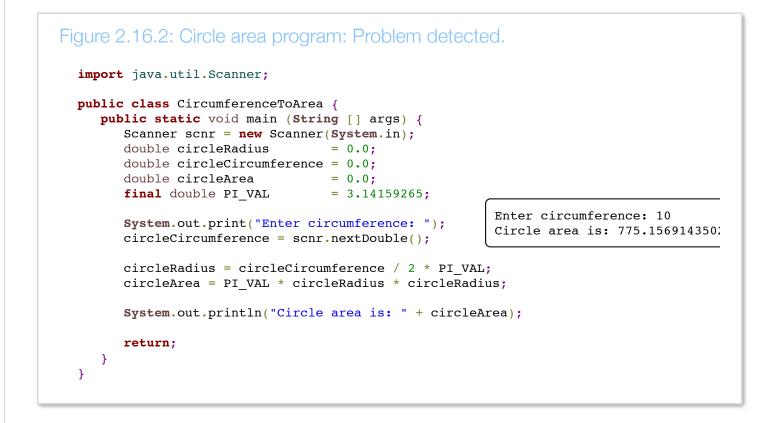
# Section 2.16 - Debugging

**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.

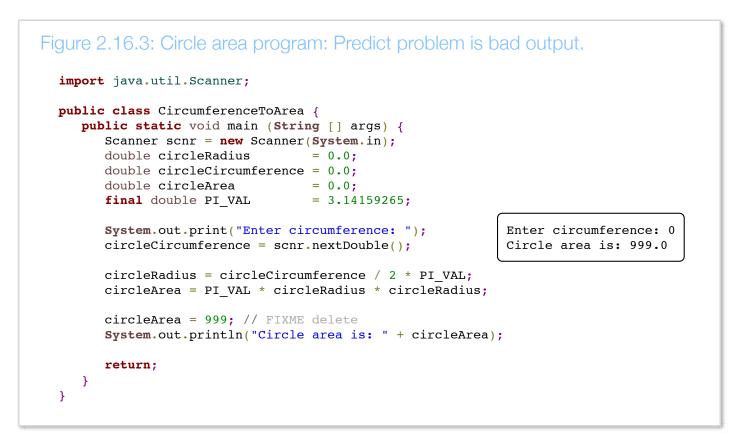


A <u>common error</u> among new programmers is to try to debug without a methodical process, instead staring at the program, or making random changes to see if the output is improved.

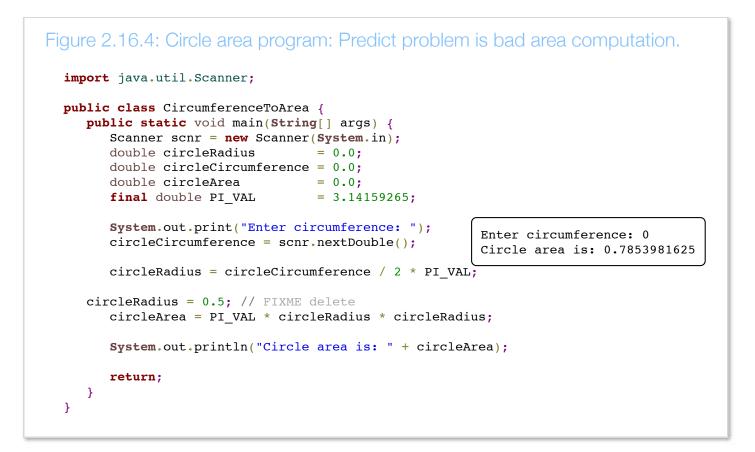
Consider a program that, given a circle's circumference, computes the circle's area. Below, the output area is clearly too large. In particular, if circumference is 10, then radius is 10 / 2 \* PI\_VAL, so about 1.6. The area is then PI\_VAL \* 1.6 \* 1.6, or about 8, but the program outputs about 775.



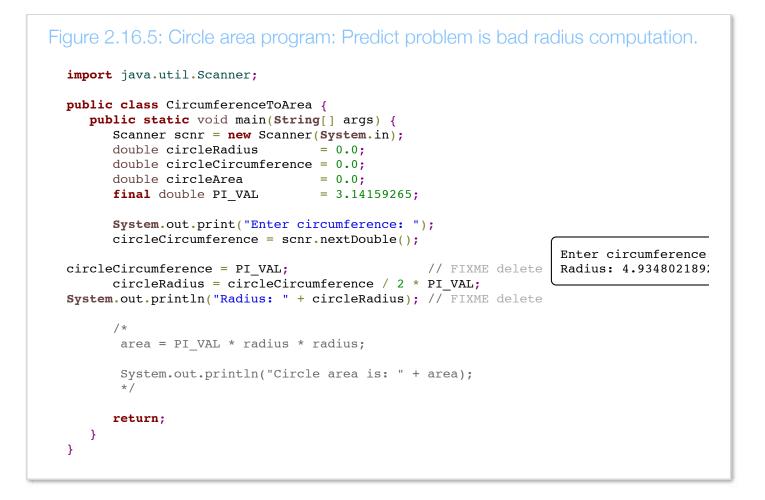
First, a programmer may predict that the problem is a bad output statement. This prediction can be tested by adding the statement area = 999;. The output statement is OK, and the predicted problem is invalidated. Note that a temporary statement commonly has a "FIXME" comment to remind the programmer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



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.



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.



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 <u>common error</u> 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 <u>common error</u> 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.



2.16.1: Debugging using a repeated two-step process.

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

```
1
                                                              10000
 2 import java.util.Scanner;
 3
 4 public class CubeVolume {
      public static void main (String [] args) {
 5
                                                                Run
         Scanner scnr = new Scanner(System.in);
 6
 7
         int sideLength = 0;
 8
         int cubeVolume = 0;
9
         System.out.println("Enter cube's side length: ");
10
         sideLength = scnr.nextInt();
11
12
13
         cubeVolume = sideLength * sideLength * sideLength;
14
15
         System.out.println("Cube's volume is: " + cubeVolu
16
17
         return;
18
      }
193
```

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

## 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.

Sample guidelines, used in this material	Yes	No (for our sample style)
<u>Whitespace</u>		
Each statement usually appears on its own line.	x = 25; y = x + 1;	x = 25; y = x + 1; if (x == 5) { y = 14; }
A blank line can separate conceptually distinct groups of statements, but related statements usually have no blank lines between them.	x = 25; y = x + 1;	x = 25; // No y = x + 1;
Most items are separated by one space (and not less or more). No space precedes an ending semicolon.	C = 25; F = ((9 * C) / 5) + 32; F = F / 2;	C=25; // No F = ((9*C)/5) + 32; // No F = F / 2 ; // No
Sub-statements are indented 3 spaces from parent statement. Tabs are not used as they may behave	<pre>if (a &lt; b) {     x = 25.</pre>	<pre>if (a &lt; b) {     x = 25; // No     y = x + 1; // No</pre>

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

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

*K&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

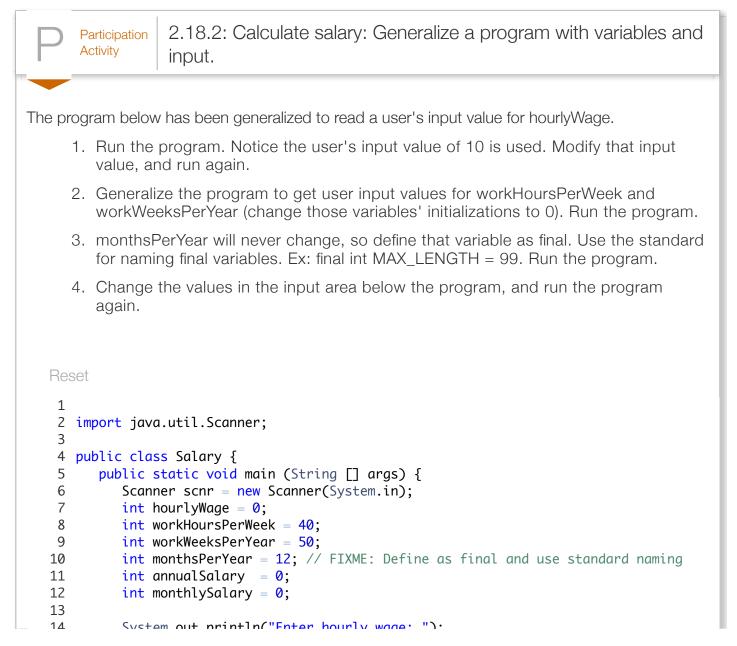
## Section 2.18 - Java example: Salary calculation with variables

Using variables in expressions, rather than numbers like 40, makes a program more general and

makes expressions more meaningful when read too.

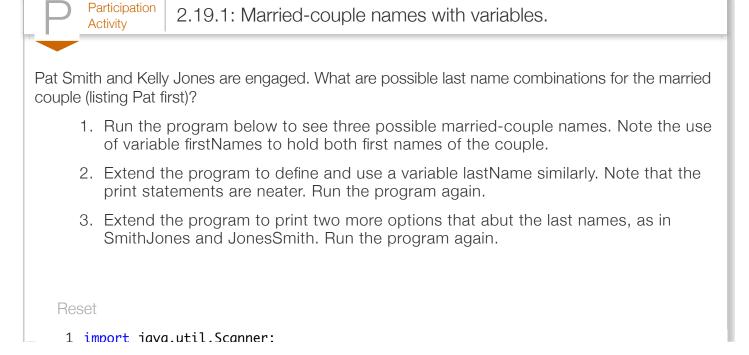
	ving program uses a variable workHoursPerWeek rather than directly using 40 in the culation expression.
,	Run the program, observe the output. Change 40 to 35 (France's work week), and run again.
2.	Generalize the program further by using a variable workWeeksPerYear . Run the program. Change 50 to 52, and run again.
3.	Introduce a variable monthlySalary, used similarly to annualSalary, to further improve program readability.
6 7 8 9 10 11 12 13	<pre>// FIXME: Define and initialize variable workWeeksPerYear, then replace the int annualSalary = 0; annualSalary = hourlyWage * workHoursPerWeek * 50; System.out.print("Annual salary is: "); System.out.println(annualSalary); System.out.print("Monthly salary is: ");</pre>
14 15 16 17	<pre>System.out.println((hourlyWage * workHoursPerWeek * 50) / 12); return; }</pre>
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.



15	<pre>bourlyWage = scnr.nextInt();</pre>
16 17	// FIXME: Get user input values for workHoursPerWeek and workWeeksPerYear
18 19	annualSalary = hourlyWaae * workHoursPerWeek * workWeeksPerYear:
10	
Run —	

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



2	······································
	<pre>public class ShowMarriedNames {</pre>
4	
5	<pre>public static void main(String[] args) {</pre>
6	<pre>Scanner scnr = new Scanner(System.in);</pre>
7	<pre>String firstName1 = "";</pre>
8	<pre>String lastName1 = "";</pre>
9	<pre>String firstName2 = "";</pre>
10	<pre>String lastName2 = "";</pre>
11	<pre>String firstNames = "";</pre>
12	// FIXME: Define lastName
13	System out println("What is the first person is first nemo2");
14 15	System.out.println("What is the first person's first name?"); firstName1 = scnr.nextLine();
16	System.out.println("What is the first person's last name?");
17	lastName1 = scnr.nextLine();
18	
	System.out.println("What is the second person's first name?"):
19	SYSTEM. OUT. DITITITI WHAT IS THE SECOND DEPSON STITIST NUME? ).
Pat	System.out.Dithth what is the second person's itrst hume? ).
	System.out.Dititting what is the second berson's itrst hume? ).
Pat	System.out.Dithth What is the second person's first hume? ).
Pat Smith Kelly	
Pat Smith	
Pat Smith Kelly	

```
Participation
                   2.19.2: Married-couple names with variables (solution).
       Activity
A solution to the above problem follows:
   Reset
    1 import java.util.Scanner;
    2
    3 public class ShowMarriedNames {
    4
    5
         public static void main(String[] args) {
             Scanner scnr = new Scanner(System.in);
    6
    7
             String firstName1 = "";
    8
             String lastName1 = "";
             String firstName2 = "";
    9
   10
             String lastName2 = "";
             String firstNames = "";
   11
                               = "";
   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
            System.out.println("What is the second person's first name?"):
Pat
Smith
Kelly
   Run
```