Review pages for the AP CS Exam

NOTE: Sections that are not considered part of the Java Subset for the AP CS exam are not included in these pages, for example Scanners.

Taken from the UW 143 Review

Building Java Programs: Ch. 1-10

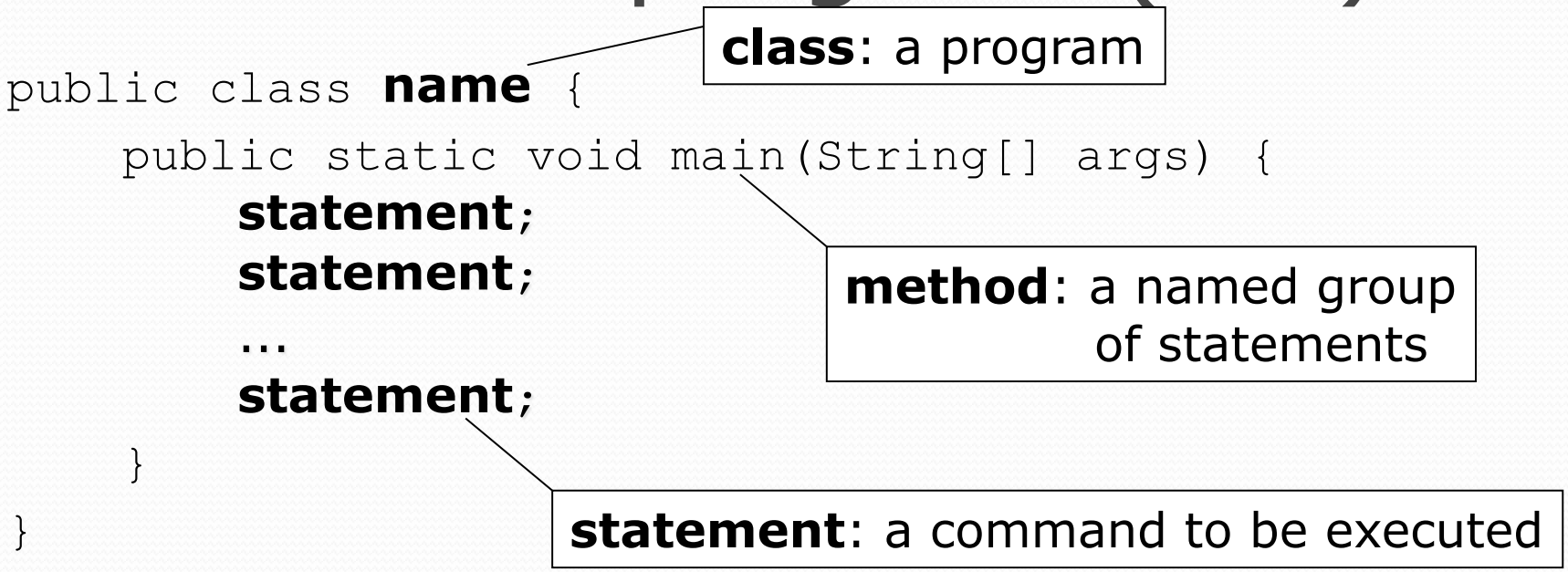
Subset of the Supplement Lesson slides from: Building Java Programs, Chapter 1-10
by Stuart Reges and Marty Stepp (http://www.buildingjavaprograms.com/) & thanks to Ms Martin.

List of Items not included in the AP CS exam, but we covered in class to build projects.

These items are not included in these pages

- `char`: Chapter 4.4
- Scanners
- Drawing Panel & Graphics
- Random Class (5.1)
- Reading Files (6.1)
- Input Tokens (6.2)
- Output to Files (6.4)
- PrintStream

A Java program (1.2)



- Every executable Java program consists of a **class**,
  - that contains a **method** named `main`,
    - that contains the **statements** (commands) to be executed.

System.out.println

- A statement that prints a line of output on the console.
  - pronounced "print-linn"
  - sometimes called a "println statement" for short

- Two ways to use `System.out.println` :
  - `System.out.println("`**`text`**`");`

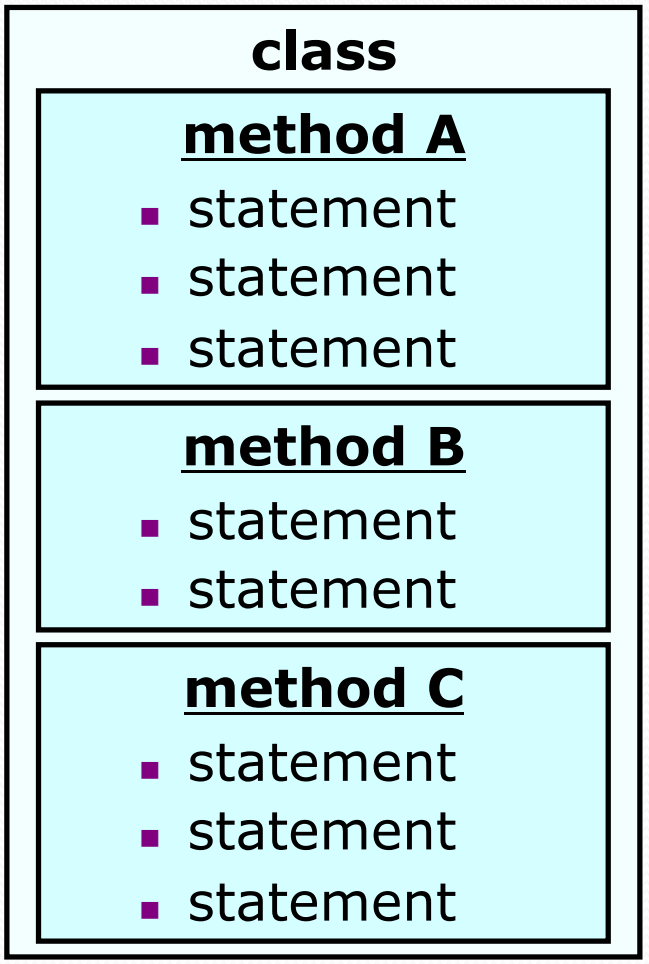
    Prints the given message as output.

  - `System.out.println();`

    Prints a blank line of output.

Static methods (1.4)

- **static method**: A named group of statements.
  - denotes the *structure* of a program
  - eliminates *redundancy* by code reuse
  - **procedural decomposition**: dividing a problem into methods
- Writing a static method is like adding a new command to Java.



Declaring a method

Gives your method a name so it can be executed

- Syntax:

```
public static void name() {
    statement;
    statement;
    ...
    statement;
}
```

- Example:

```
public static void printWarning() {
    System.out.println("This product causes cancer");
    System.out.println("in lab rats and humans.");
}
```

Calling a method

Executes the method's code

- Syntax:

```
name();
```

  - You can call the same method many times if you like.

- Example:

```
printWarning();
```

  - Output:

```
This product causes cancer
in lab rats and humans.
```

Control flow

- When a method is called, the program's execution...
  - "jumps" into that method, executing its statements, then
  - "jumps" back to the point where the method was called.

```
public class MethodsExample {
    public static void main(String[] args) {
        message1();

        message2();

        System.out.println("
    }

    ...
}
```

```
public static void message1() {
    System.out.println("This is message1.");
}
```

```
public static void message2() {
    System.out.println("This is message2.");
    message1();

    System.out.println("Done with message2.");
}
```

```
public static void message1() {
    System.out.println("This is message1.");
}
```

Java's primitive types (2.1)

- **primitive types**: 8 simple types for numbers, text, etc.
  - Java also has **object types**, which we'll talk about later

Name	Description	Examples
`int`	integers	`42, -3, 0, 926394`
`double`	real numbers	`3.1, -0.25, 9.4e3`
`char`	single text characters	`'a', 'X', '?', '\n'`
`boolean`	logical values	`true, false`

  - Why does Java distinguish integers vs. real numbers?

Expressions

- **expression**: A value or operation that computes a value.
  - Examples:

```
1 + 4 * 5
(7 + 2) * 6 / 3
42
```

  - The simplest expression is a *literal value*.
  - A complex expression can use operators and parentheses.

Integer division with /

- When we divide integers, the quotient is also an integer.
  - `14 / 4` is `3`, not `3.5`

```
   3
4 ) 14
   12
    2
```

```
   4
10 ) 45
   40
    5
```

```
   52
27 ) 1425
   135
    75
    54
    21
```

- More examples:
  - `32 / 5` is `6`
  - `84 / 10` is `8`
  - `156 / 100` is `1`
  - Dividing by 0 causes an error when your program runs.

Integer remainder with %

- The % operator computes the remainder from integer division.
  - `14 % 4` is `2`
  - `218 % 5` is `3`

```
    3
4 ) 14
    12
    2
```

```
    43
5 ) 218
    20
    18
    15
    3
```

What is the result?

```
45 % 6
2 % 2
8 % 20
11 % 0
```

- Applications of % operator:
  - Obtain last digit of a number: `230857 % 10` is `7`
  - Obtain last 4 digits: `658236489 % 10000` is `6489`
  - See whether a number is odd: `7 % 2` is `1`, `42 % 2` is `0`

Precedence

- **precedence**: Order in which operators are evaluated.
  - Generally operators evaluate left-to-right.
    `1 - 2 - 3` is `(1 - 2) - 3` which is `-4`
  - But `*/%` have a higher level of precedence than `+-`

```
1 + 3 * 4                is 13

6 + 8 / 2 * 3
6 +   4   * 3
6 +     12               is 18
```

  - Parentheses can force a certain order of evaluation:
    `(1 + 3) * 4` is `16`
  - Spacing does not affect order of evaluation
    `1+3 * 4-2` is `11`

String concatenation

- **string concatenation**: Using + between a string and another value to make a longer string.

```
"hello" + 42     is "hello42"
1 + "abc" + 2    is "1abc2"
"abc" + 1 + 2    is "abc12"
1 + 2 + "abc"    is "3abc"
"abc" + 9 * 3    is "abc27"
"1" + 1          is "11"
4 - 1 + "abc"    is "3abc"
```

- Use + to print a string and an expression's value together.
  - `System.out.println("Grade: " + (95.1 + 71.9) / 2);`
  - Output: `Grade: 83.5`

Variables (2.2)

- **variable**: A piece of the computer's memory that is given a name and type, and can store a value.

- A variable can be declared/initialized in one statement.

- Syntax:

  type name = **value**;

  - `double myGPA = 3.95;`

  - `int x = (11 % 3) + 12;`

Type casting

- **type cast**: A conversion from one type to another.
  - To promote an `int` into a `double` to get exact division from `/`
  - To truncate a `double` from a real number to an integer

- Syntax:

  `(`**type**`)` **expression**

  Examples:

```
double result = (double) 19 / 5;      // 3.8
int result2 = (int) result;           // 3
int x = (int) Math.pow(10, 3);        // 1000
```

Increment and decrement

shortcuts to increase or decrease a variable's value by 1

Shorthand

```
variable++;
variable--;
```

Equivalent longer version

```
variable = variable + 1;
variable = variable - 1;
```

```
int x = 2;
x++;

double gpa = 2.5;
gpa--;
```

```
// x = x + 1;
// x now stores 3

// gpa = gpa - 1;
// gpa now stores 1.5
```

`Integer` methods

`intValue(`**value**`)`	Returns value as an int
`Integer.MIN_VALUE`	A constant holding the minimum value an int can have
`Integer.MAX_VALUE`	A constant holding the maximum value an int can have
`Integer(`**value**`)`	Constructs a newly allocated Integer object that represents the specified int value.

Part of the java.lang.Integer NOTE: these are the only List methods used in the AP CS Exam.

Modify-and-assign operators

shortcuts (NOT REQUIRED for AP CS A)

Shorthand	Equivalent longer version
variable += **value**;	**variable** = **variable** + **value**;
variable -= **value**;	**variable** = **variable** - **value**;
variable *= **value**;	**variable** = **variable** * **value**;
variable /= **value**;	**variable** = **variable** / **value**;
variable %= **value**;	**variable** = **variable** % **value**;

```
x += 3;                    // x = x + 3;
gpa -= 0.5;                // gpa = gpa - 0.5;
number *= 2;               // number = number * 2;
```

`for` loops (2.3)

```
for (initialization; test; update) {
    statement;
    statement;
    ...
    statement;
}
```

header

body

- Perform **initialization** once.
- Repeat the following:
  - Check if the **test** is true. If not, stop.
  - Execute the **statement**s.
  - Perform the **update**.

System.out.print

- Prints without moving to a new line
  - allows you to print partial messages on the same line

```
int highestTemp = 5;
for (int i = -3; i <= highestTemp / 2; i++) {
    System.out.print((i * 1.8 + 32) + "  ");
}
```

  - Output:

```
26.6  28.4  30.2  32.0  33.8  35.6
```

Nested loops

- **nested loop**: A loop placed inside another loop.

```
for (int i = 1; i <= 4; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.print((i * j) + "\t");
    }
    System.out.println();  // to end the line
}
```

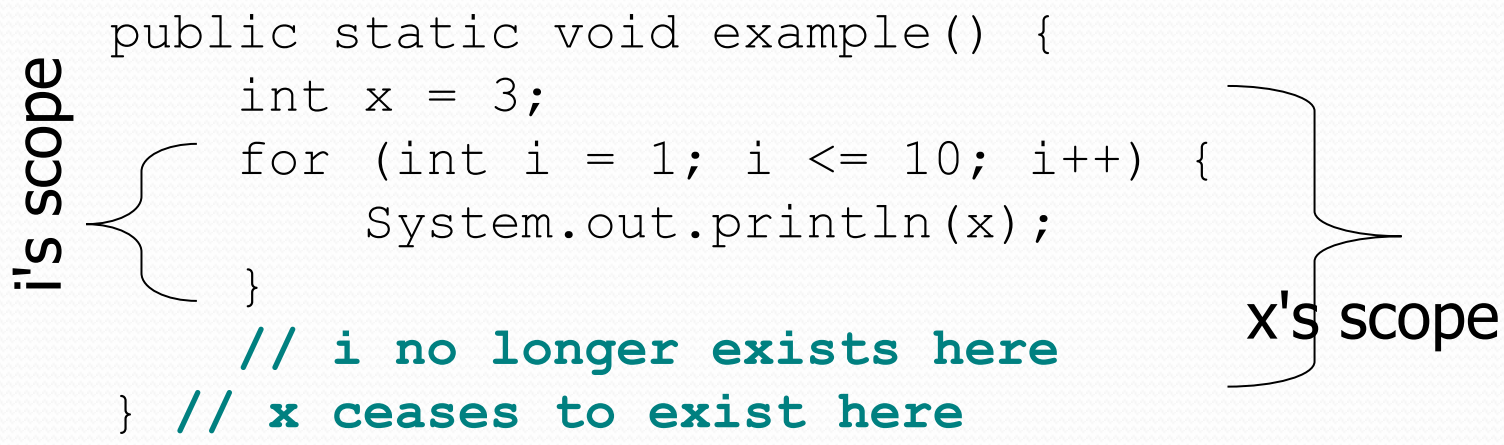
- Output:

```
1    2    3    4    5
2    4    6    8    10
3    6    9    12   15
4    8    12   16   20
```

- Statements in the outer loop's body are executed 4 times.
  - The inner loop prints 5 numbers each time it is run.

Variable scope

- **scope**: The part of a program where a variable exists.
  - From its declaration to the end of the `{ }` braces
    - A variable declared in a `for` loop exists only in that loop.
    - A variable declared in a method exists only in that method.




```
public static void example() {
    int x = 3;
    for (int i = 1; i <= 10; i++) {
        System.out.println(x);
    }
    // i no longer exists here
} // x ceases to exist here
```

Class constants (2.4)

- **class constant**: A value visible to the whole program.
  - value can only be set at declaration
  - value can't be changed while the program is running

- Syntax:

```
public static final type name = value;
```

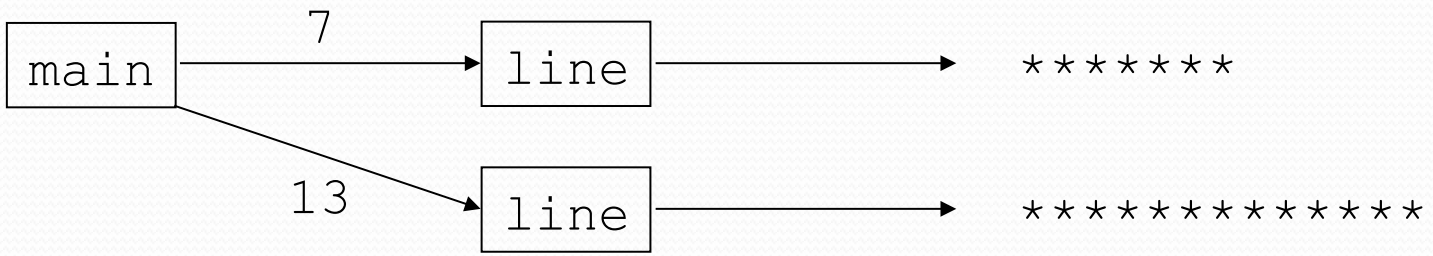
  - name is usually in ALL_UPPER_CASE

  - Examples:

```
public static final int DAYS_IN_WEEK = 7;
public static final double INTEREST_RATE = 3.5;
public static final int SSN = 658234569;
```

Parameters (3.1)

- **parameter**: A value passed to a method by its caller.
  - Instead of `lineOf7`, `lineOf13`, write `line` to draw any length.
    - When *declaring* the method, we will state that it requires a parameter for the number of stars.
    - When *calling* the method, we will specify how many stars to draw.



Passing parameters

- Declaration:

```
public static void name (type name, ..., type name) {
    statement(s);
}
```

- Call:

```
methodName (value, value, ..., value);
```

- Example:

```
public static void main(String[] args) {
    sayPassword(42);        // The password is: 42
    sayPassword(12345);     // The password is: 12345
}

public static void sayPassword(int code) {
    System.out.println("The password is: " + code);
}
```

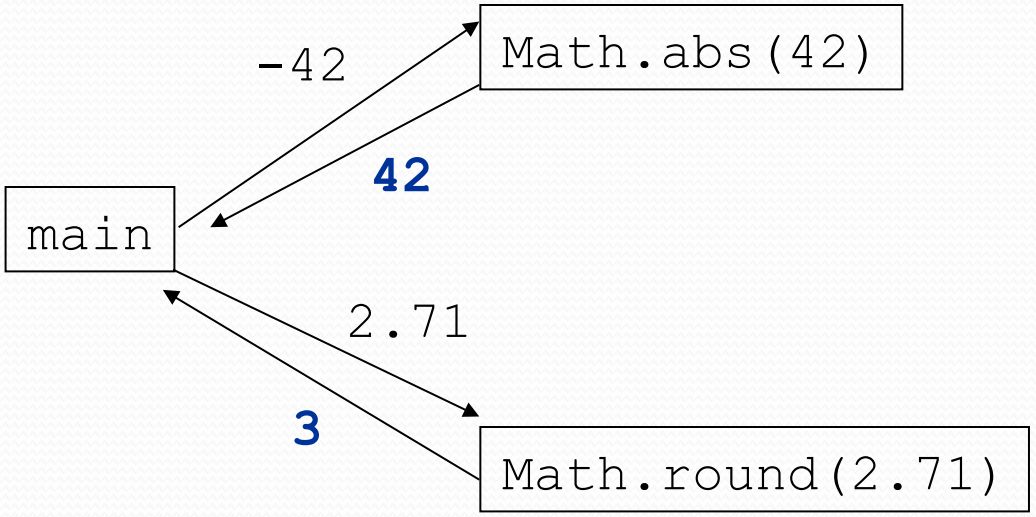
Java's `Math` class (3.2)

Method name	Description
`Math.abs(int` *value*`)`	returns int absolute value
`Math.abs(double` *value*`)`	returns double absolute value
`Math.pow(`*base*`, `*exp*`)`	*base* to the *exp* power
`Math.sqrt(`*value*`)`	square root
`Math.random()`	random `double` between 0 and 1

These are the only Math Methods required for the AP CS, others can be used for the Free Response Questions (FRQ), but will not be in the Multiple Choice.

Return (3.2)

- **return**: To send out a value as the result of a method.
  - The opposite of a parameter:
    - Parameters send information *in* from the caller to the method.
    - Return values send information *out* from a method to its caller.



Returning a value

```
public static type name(parameters) {
    statements;
    ...
    return expression;
}
```

- Example:

```
// Returns the slope of the line between the given points.
public static double slope(int x1, int y1, int x2, int y2) {
    double dy = y2 - y1;
    double dx = x2 - x1;
    return dy / dx;
}
```

Method Overloading (3.1)

- **method overloading**: The ability to define two different or more different methods with the same name but different number and/or type of parameters.

```
public static void drawBox() { // no parameters
  // has code that creates a standard sized box
  ...
}
public static void drawBox(int height, int width) {
  // code that draws the box based on the height and
  // width parameter values
  ...
}
```

Which method used is based on how it is called:

```
drawBox();  // uses the first drawBox method
drawBox(10,20);  // uses second drawBox method
```

Strings (3.3)

- **string**: An object storing a sequence of text characters.

```
String name = "text";
String name = expression;
```

  - Characters of a string are numbered with 0-based *indexes*:

```
String name = "P. Diddy";
```

index	0	1	2	3	4	5	6	7
char	P	.		D	i	d	d	y

  - The first character's index is always 0
  - The last character's index is 1 less than the string's length
  - The individual characters are values of type `char`

`String` methods

Method name	Description
`indexOf(`**`str`**`)`	index where the start of the given string appears in this string (-1 if it is not there)
`length()`	number of characters in this string
`substring(`**`index1`**`,` **`index2`**`)` or `substring(`**`index1`**`)`	the characters in this string from *index1* (inclusive) to *index2* (exclusive); if *index2* omitted, grabs till end of string
`compareTo(String other)`	returns <0 if this is less than other returns 0 if this is equal to other returns >0 if this is greater than other

- These methods are called using the dot notation:

```
String gangsta = "Dr. Dre";
System.out.println(gangsta.length());    // 7
```

Note: These are the only String methods required for the AP CS...

The `equals` method

- Objects are compared using a method named `equals`.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney") {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

  - Technically this is a method that returns a value of type `boolean`, the type used in logical tests.

Cumulative sum (4.1)

- A loop that adds the numbers from 1-1000:

```
int sum = 0;
for (int i = 1; i <= 1000; i++) {
    sum = sum + i;
}
System.out.println("The sum is " + sum);
```

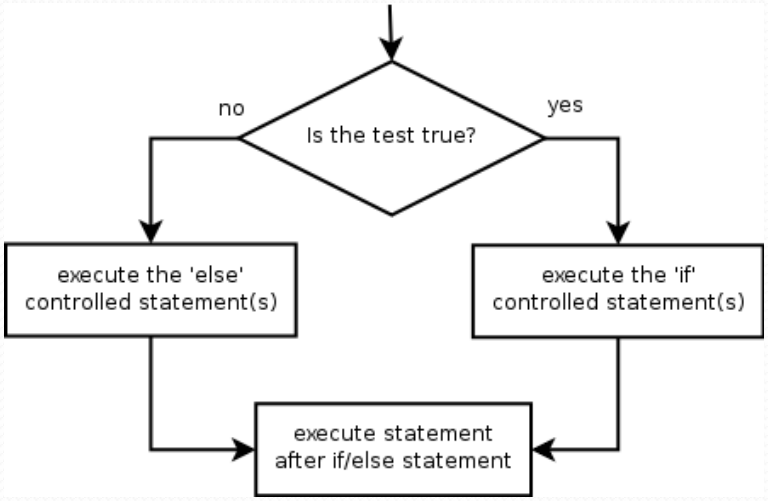
Key idea:

- Cumulative sum variables must be declared *outside* the loops that update them, so that they will exist after the loop.

if/else (4.2)

Executes one block if a test is true, another if false

```
if (test) {
    statement(s);
} else {
    statement(s);
}
```



- Example:

```
double gpa = console.nextDouble();
if (gpa >= 2.0) {
    System.out.println("Welcome to Mars University!");
} else {
    System.out.println("Application denied.");
}
```

Relational expressions

- A **test** in an `if` is the same as in a `for` loop.

```
for (int i = 1; i <= 10; i++) { ...
if (i <= 10) { ...
```

  - These are `boolean` expressions, seen in Ch. 5.

- Tests use *relational operators*:

Operator	Meaning	Example	Value
==	equals	`1 + 1 == 2`	`true`
!=	does not equal	`3.2 != 2.5`	`true`
<	less than	`10 < 5`	`false`
>	greater than	`10 > 5`	`true`
<=	less than or equal to	`126 <= 100`	`false`
>=	greater than or equal to	`5.0 >= 5.0`	`true`

Logical operators: `&&`, `||`, `!`

- Conditions can be combined using *logical operators*:

Operator	Description	Example	Result
`&&`	and	`(2 == 3) && (-1 < 5)`	`false`
`||`	or	`(2 == 3) || (-1 < 5)`	`true`
`!`	not	`!(2 == 3)`	`true`

- "Truth tables" for each, used with logical values *p* and *q*:

p	q	p && q	p || q
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

p	!p
true	false
false	true

Type `boolean` (5.2)

- **`boolean`**: A logical type whose values are `true` and `false`.
  - A **test** in an `if`, `for`, or `while` is a `boolean` expression.
  - You can create `boolean` variables, pass `boolean` parameters, return `boolean` values from methods, ...

```
boolean minor = (age < 21);
boolean expensive = iPhonePrice > 200.00;
boolean iLoveCS = true;

if (minor) {
    System.out.println("Can't purchase alcohol!");
}
if (iLoveCS || !expensive) {
    System.out.println("Buying an iPhone");
}
```

De Morgan's Law

- **De Morgan's Law**:
  Rules used to *negate* or *reverse* boolean expressions.
  - Useful when you want the opposite of a known boolean test.

Original Expression	Negated Expression	Alternative
`a && b`	`!a || !b`	`!(a && b)`
`a || b`	`!a && !b`	`!(a || b)`

  - Example:

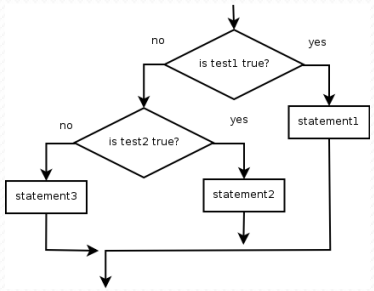
Original Code	Negated Code

`if (x == 7 && y > 3) {`
`  ...`
`}`
	`if (x != 7 || y <= 3) {`
	`  ...`
	`}`

`if/else` Structures

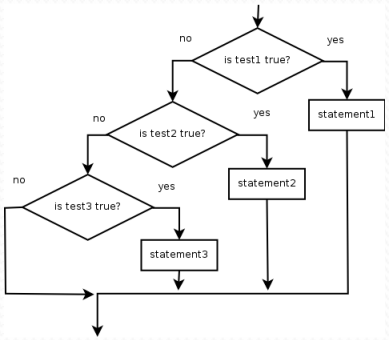
- Exactly 1 path: (mutually exclusive)

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```



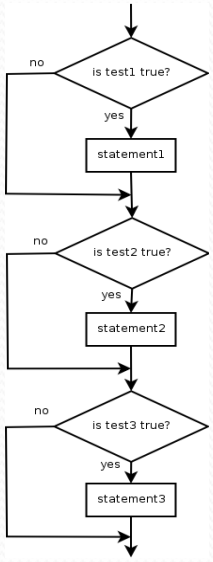
- 0 or 1 path:

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else if (test) {
    statement(s);
}
```



- 0, 1, or many paths: (independent tests, not exclusive)

```
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
```

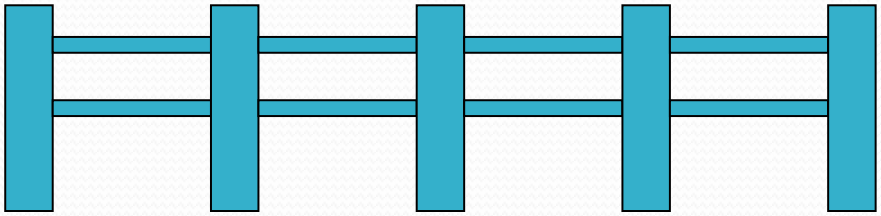


Fencepost loops (4.1)

- **fencepost problem**: When we want to repeat two tasks, one of them *n* times, another *n-1* or *n+1* times.
  - Add a statement outside the loop to place the initial "post."
  - Also called a *fencepost loop* or a "loop-and-a-half" solution.

  - Algorithm template:

place a post.
for (length of fence ***- 1****) {*
    place some wire.
    place a post.
}



Fencepost method solution

- Write a method `printNumbers` that prints each number from 1 to a given maximum, separated by commas.

  For example, the call:

```
printNumbers(5);
```

  should print:

```
1, 2, 3, 4, 5
```

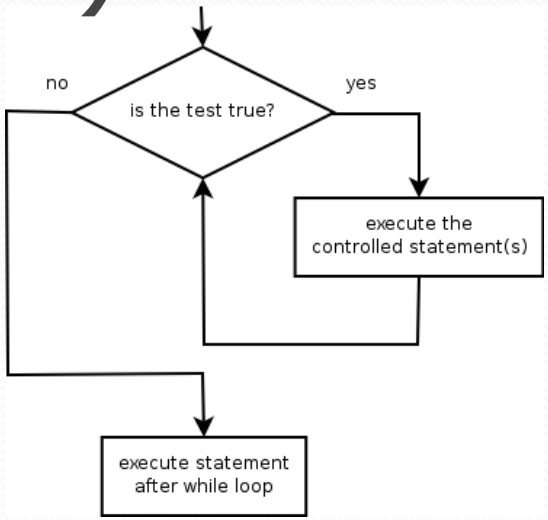
- Solution:

```
public static void printNumbers(int max) {
    System.out.print(1);
    for (int i = 2; i <= max; i++) {
        System.out.print(", " + i);
    }
    System.out.println();     // to end the line
}
```

`while` loops (5.1)

- **`while` loop**: Repeatedly executes its body as long as a logical test is true.

```
while (test) {
    statement(s);
}
```



- Example:

```
int num = 1;                          // initialization
while (num <= 200) {                  // test
    System.out.print(num + " ");
    num = num * 2;                    // update
}
```

  - OUTPUT:

```
1 2 4 8 16 32 64 128
```

`do/while` loops (5.4)

- **`do/while` loop**: Executes statements repeatedly while a condition is `true`, testing it at the *end* of each repetition.

```
do {
    statement(s);
} while (test);
```

  - Example:

```
// prompt until the user gets the right password
String phrase;
do {
    System.out.print("Password: ");
    phrase = console.next();
} while (!phrase.equals("abracadabra"));
```

"Boolean Zen"

- Students new to `boolean` often test if a result is `true`:

```
if (bothOdd(7, 13) == true) {     // bad
    ...
}
```

- But this is unnecessary and redundant. Preferred:

```
if (bothOdd(7, 13)) {             // good
    ...
}
```

- A similar pattern can be used for a `false` test:

```
if (bothOdd(7, 13) == false) {    // bad
if (!bothOdd(7, 13)) {            // good
```

"Boolean Zen", part 2

- Methods that return `boolean` often have an `if/else` that returns `true` or `false`:

```
public static boolean bothOdd(int n1, int n2) {
    if (n1 % 2 != 0 && n2 % 2 != 0) {
        return true;
    } else {
        return false;
    }
}
```

- Observation: The `if/else` is unnecessary.
  - Our logical test is itself a `boolean` value; so return that!

```
public static boolean bothOdd(int n1, int n2) {
    return (n1 % 2 != 0 && n2 % 2 != 0);
}
```

The `throws` clause

- **`throws` clause**: Keywords on a method's header that state that it may generate an exception.

- Syntax:

```
public static type name(params) throws type {
```

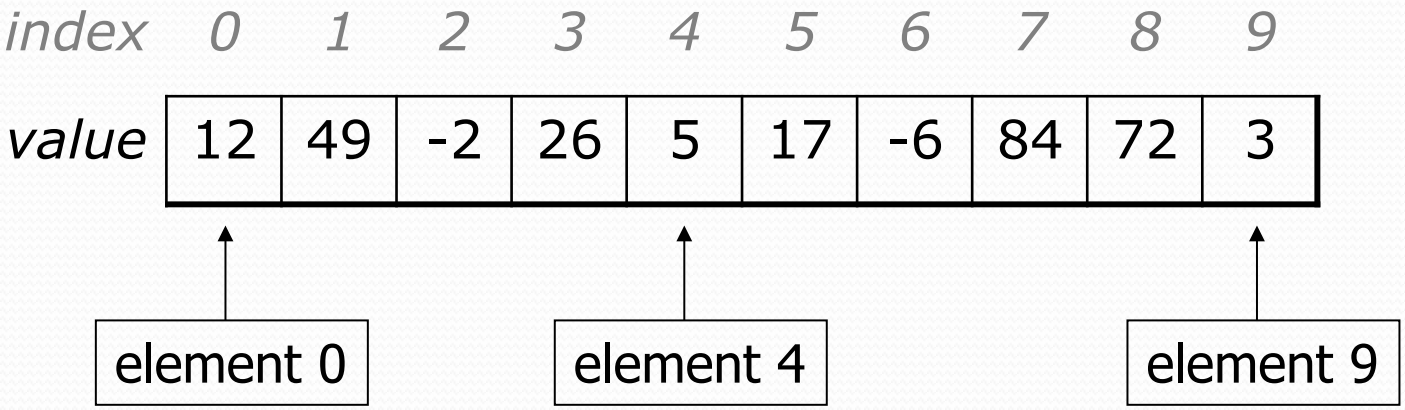
  - Example:

```
public class ReadFile {
    public static void main(String[] args)
            throws FileNotFoundException {
```

  - Like saying, *"I hereby announce that this method might throw an exception, and I accept the consequences if it happens."*

Arrays (7.1)

- **array**: object that stores many values of the same type.
  - **element**: One value in an array.
  - **index**: A 0-based integer to access an element from an array.

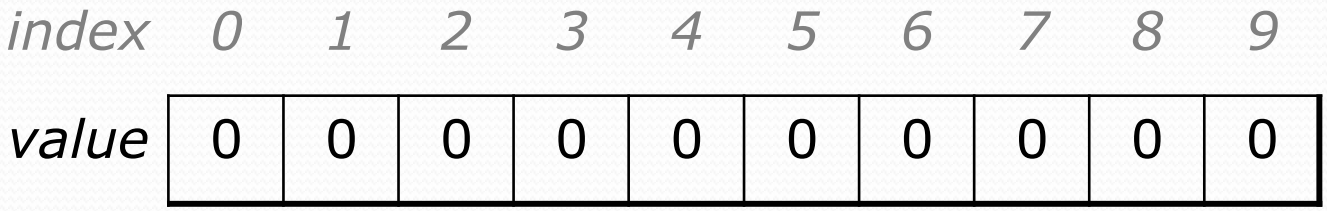


Array declaration

type`[]` **name** `=` `new` **type**`[`**length**`];`

- Example:

```
int[] numbers = new int[10];
```

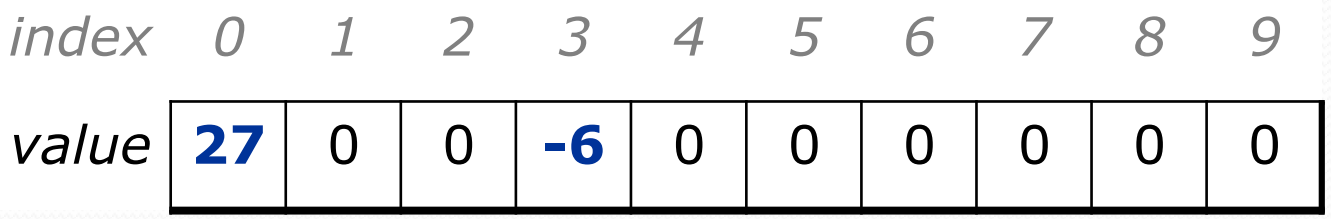


Accessing elements

```
name[index]                    // access
name[index] = value;           // modify
```

- Example:

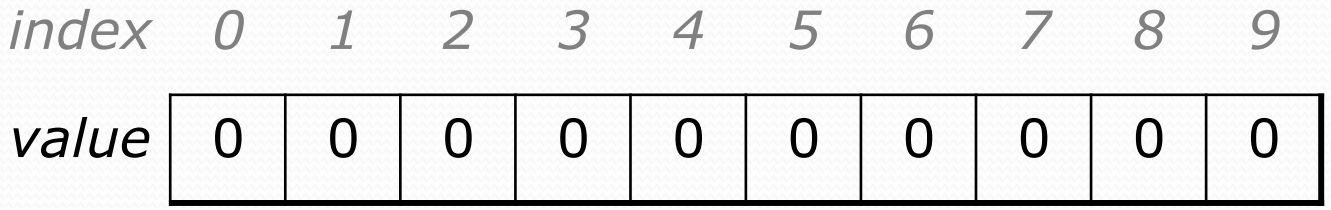
```
numbers[0] = 27;
numbers[3] = -6;

System.out.println(numbers[0]);
if (numbers[3] < 0) {
    System.out.println("Element 3 is negative.");
}
```



Out-of-bounds

- Legal indexes: between **0** and the **array's length - 1**.
  - Reading or writing any index outside this range will throw an `ArrayIndexOutOfBoundsException`.
- Example:

```
int[] data = new int[10];
System.out.println(data[0]);        // okay
System.out.println(data[9]);        // okay
System.out.println(data[-1]);       // exception
System.out.println(data[10]);       // exception
```



The `length` field

- An array's `length` field stores its number of elements.

  name`.length`

```
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
// output: 0 2 4 6 8 10 12 14
```

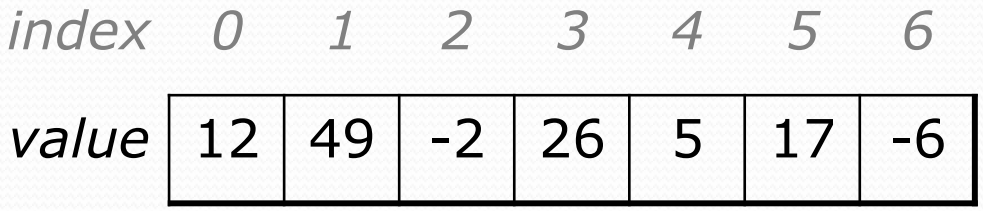
  - It does not use parentheses like a String's `.length()`.

Quick array initialization

type`[]` **name** `=` `{`**value**`,` **value**`,` … **value**`};`

- Example:

```
int[] numbers = {12, 49, -2, 26, 5, 17, -6};
```



- Useful when you know what the array's elements will be.
- The compiler figures out the size by counting the values.

The `Arrays` class

- Class `Arrays` in package `java.util` has useful static methods for manipulating arrays:

Method name	Description
`binarySearch(`**array**`, `**value**`)`	returns the index of the given value in a sorted array (< 0 if not found)
`equals(`**array1**`, `**array2**`)`	returns `true` if the two arrays contain the same elements in the same order
`fill(`**array**`, `**value**`)`	sets every element in the array to have the given value
`sort(`**array**`)`	arranges the elements in the array into ascending order
`toString(`**array**`)`	returns a string representing the array, such as `"[10, 30, 17]"`

Arrays as parameters

- Declaration:

```
public static type methodName(type[] name) {
```

  - Example:

```
public static double average(int[] numbers) {
    ...
}
```

- Call:

```
methodName(arrayName);
```

  - Example:

```
int[] scores = {13, 17, 12, 15, 11};
double avg = average(scores);
```

Arrays as return

- Declaring:

```
public static type[] methodName(parameters) {
```

  - Example:

```
public static int[] countDigits(int n) {
    int[] counts = new int[10];
    ...
    return counts;
}
```

- Calling:

```
type[] name = methodName(parameters);
```

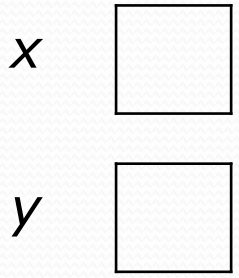
  - Example:

```
public static void main(String[] args) {
    int[] tally = countDigits(229231007);
    System.out.println(Arrays.toString(tally));
}
```

Value semantics (primitives)

- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

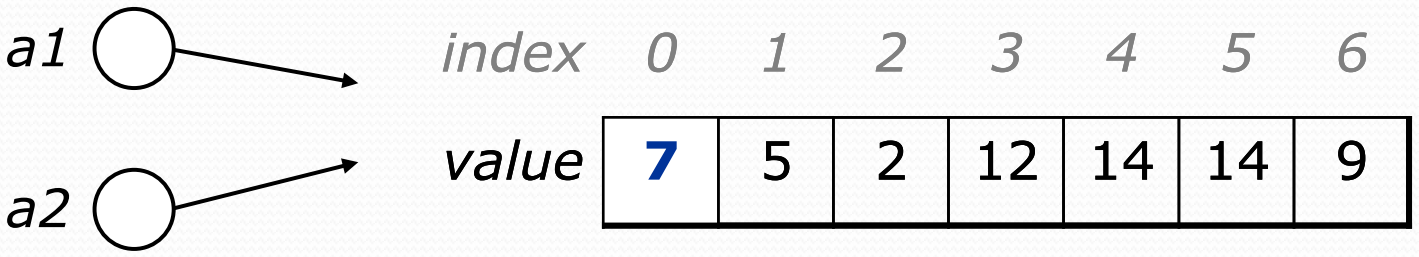
```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```



Reference semantics (objects)

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one reference variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable *will* affect others.

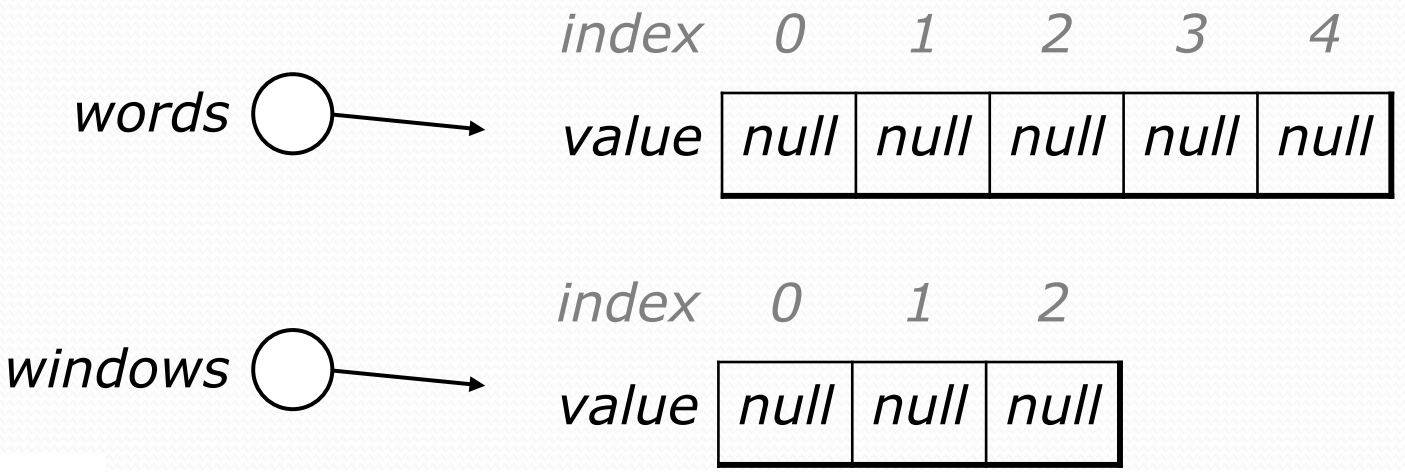
```
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1;      // refer to same array as a1
a2[0] = 7;
System.out.println(a1[0]);   // 7
```



Null

- **`null` :** A reference that does not refer to any object.
  - Fields of an object that refer to objects are initialized to `null`.
  - The elements of an array of objects are initialized to `null`.

```
String[] words = new String[5];
DrawingPanel[] windows = new DrawingPanel[3];
```



Null pointer exception

- **dereference**: To access data or methods of an object with the dot notation, such as `s.length()`.
  - It is illegal to dereference `null` (causes an exception).
  - `null` is not any object, so it has no methods or data.

```
String[] words = new String[5];
System.out.println("word is: " + words[0]);
words[0] = words[0].toUpperCase();
```

Output:

```
word is: null
Exception in thread "main"
java.lang.NullPointerException
        at Example.main(Example.java:8)
```

Classes and objects (8.1)

- **class**: A program entity that represents either:
  1. A program / module, or
  2. **A template for a new type of objects.**

  - The `DrawingPanel` class is a template for creating `DrawingPanel` objects.

- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

Fields (8.2)

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.
  - **encapsulation**: Declaring fields `private` to hide their data. With few exceptions ALWAYS Make fields `private` in Objects.

- Declaration syntax:

```
private type name;
```

  - Example:

```
public class Student {
    private String name;     // each object now has
    private double gpa;      // a name and gpa field
}
```

Instance methods

- **instance method**: One that exists inside each object of a class and defines behavior of that object.

```
public type name(parameters) {
    statements;
}
```

  - same syntax as static methods, but without `static` keyword

  Example:

```
public void shout() {
    System.out.println("HELLO THERE!");
}
```

A `Point` class

```
public class Point {
    private int x;
    private int y;

    // Changes the location of this Point object.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }
}
```

- Each `Point` object contains data fields named `x` and `y`.
- Each `Point` object contains a method named `draw` that draws that point at its current `x`/`y` position.

The implicit parameter

- **implicit parameter**:
  The object on which an instance method is called.
  - During the call `p1.draw(g);`
    the object referred to by `p1` is the implicit parameter.
  - During the call `p2.draw(g);`
    the object referred to by `p2` is the implicit parameter.
  - The instance method can refer to that object's fields.
    - We say that it executes in the *context* of a particular object.
    - `draw` can refer to the `x` and `y` of the object it was called on.

Kinds of methods

- Instance methods take advantage of an object's state.
  - Some methods allow clients to access/modify its state.

- **accessor**: A method that lets clients examine object state.
  - Example: A `distanceFromOrigin` method that tells how far a `Point` is away from (0, 0).
  - Accessors often have a non-`void` return type.

- **mutator**: A method that modifies an object's state.
  - Example: A `translate` method that shifts the position of a `Point` by a given amount.

Constructors (8.4)

- **constructor**: Initializes the state of new objects.

```
public type(parameters) {
    statements;
}
```

  - Example:

```
public Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
}
```

  - runs when the client uses the `new` keyword
  - does not specify a return type; implicitly returns a new object
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to 0.

`toString` method (8.6)

- tells Java how to convert an object into a `String`

```
public String toString() {
    code that returns a suitable String;
}
```

  - Example:

```
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

- called when an object is printed/concatenated to a `String`:

```
Point p1 = new Point(7, 2);
System.out.println("p1: " + p1);
```

- Every class has a `toString`, even if it isn't in your code.
  - Default is class's name and a hex number: `Point@9e8c34`

`this` keyword (8.7)

- **`this`** : A reference to the implicit parameter.
  - *implicit parameter:* object on which a method is called

- Syntax for using `this`:
  - To refer to a field:

    `this.`**`field`**

  - To call a method:

    `this.`**`method`**`(`**`parameters`**`);`

  - To call a constructor from another constructor:

    `this(`**`parameters`**`);`

Static methods

- **static method**: Part of a class, not part of an object.
  - shared by all objects of that class
  - good for code related to a class but not to each object's state
  - does not understand the *implicit parameter*, `this`; therefore, cannot access an object's fields directly
  - if `public`, can be called from inside or outside the class

- Declaration syntax:

```
public static type name(parameters) {
    statements;
}
```

Inheritance (9.1)

- **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes

- One class can *extend* another, absorbing its data/behavior.
  - **superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass

Inheritance syntax (9.1)

```
public class name extends superclass {
```

- Example:

```
public class Secretary extends Employee {
    ...
}
```

- By extending `Employee`, each `Secretary` object now:
  - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
  - can be treated as an `Employee` by client code (seen later)

Overriding methods (9.1)

- **override**: To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```
public class Secretary extends Employee {
    // overrides getVacationForm in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```

`super` keyword (9.3)

- Subclasses can call overridden methods with `super`

  `super.`**`method`**`(`**`parameters`**`)`

  - Example:

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.0;
    }
    ...
}
```

Abstract Class

- **Abstract Class:** A Java Class that cannot be instantiated, but that instead serves as a superclass to hold common code and declare abstract behavior (methods)

- Abstract classes are useful when a superclass doesn't correspond to a real "thing" but more of an idea
  - abstract classes can't be instantiated (no new…)
  - they can have fields
  - they can have concrete methods
  - they generally have abstract methods

```
public abstract class name {
  private <type> name; // can have fields
    //can have abstract and/or concrete methods...
  public type name(type name, ..., type name);
  public type name(type name, ..., type name){
    ... // statements of method
  }
}
```

Interfaces

- **interface**: A list of abstract methods that a class can implement.
  - Interfaces give you an is-a relationship *without* code sharing.
- **abstract method**: A header without an implementation.
  - The actual body is not specified, to allow/force different classes to implement the behavior in its own way.

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
}
```

Example:

```
public interface Vehicle {
    public double speed();
    public void setDirection(int direction);
}
```

Implementing an interface

```
public class name implements interface {
    ...
}
```

- Example:

```
public class Bicycle implements Vehicle {
    ...
}
```

- A class can declare that it *implements* an interface.
  - This means the class must contain each of the abstract methods in that interface. (Otherwise, it will not compile.)

An Abstract Class can implement an interface

```
public abstract class name implements interface_name {
  ...
}
```

Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
  - Example: `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.

- A variable of type *T* can hold an object of any subclass of *T*.

```
Employee ed = new LegalSecretary();
```

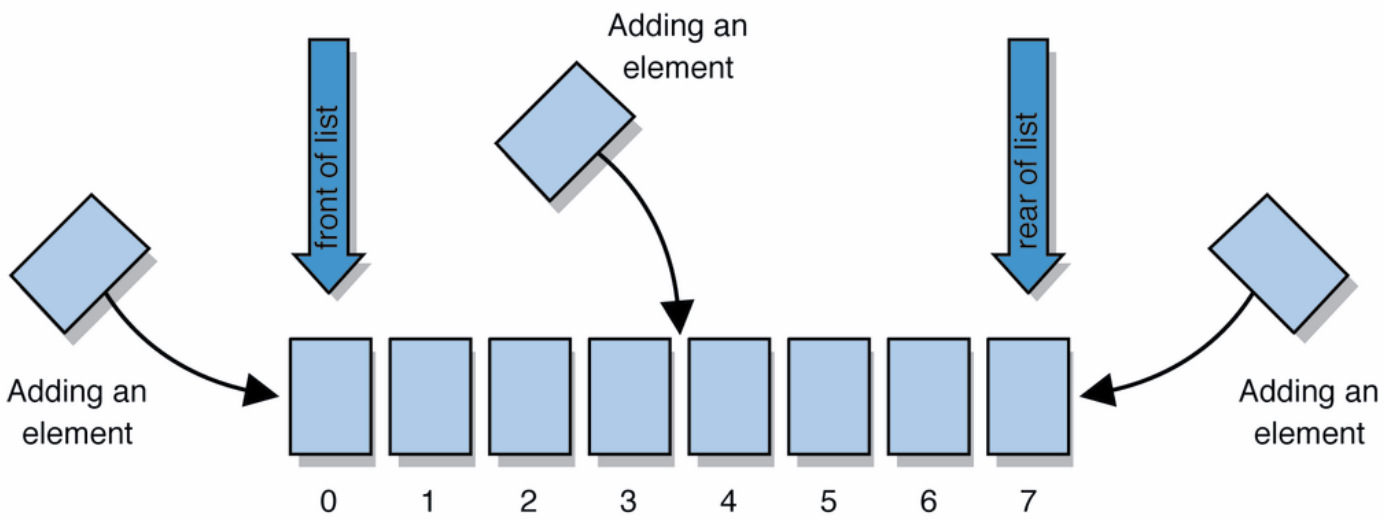
  - You can call any methods from `Employee` on `ed`.
  - You can *not* call any methods specific to `LegalSecretary`.

- When a method is called, it behaves as a `LegalSecretary`.

```
System.out.println(ed.getSalary());          // 55000.0
System.out.println(ed.getVacationForm());    // pink
```

Lists (10.1)

- **list**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **`ArrayList`** object



Idea of a list

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

```
[]
```

- You can add items to the list.
  - The default behavior is to add to the end of the list.

```
[hello, ABC, goodbye, okay]
```

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

Type Parameters (Generics)

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `<` and `>`.
  - This is called a *type parameter* or a *generic* class.
  - Allows the same `ArrayList` class to store lists of different types.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

`ArrayList` methods (10.1)

`add(`**value**`)`	appends value at end of list (returns true boolean)
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(int` **index**`)`	removes & returns value (Object) at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value & returns the previous Object, if needed
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

class java.util.ArrayList<E> implements java.util.List<E>

These are the only List methods used in the AP CS Exam.