Java Review

Variables

- **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;`

x	14

myGPA	3.95

Java's primitive types

- **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?

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	Equivalent longer version
variable`++;`	**variable** `=` **variable** `+ 1;`
variable`--;`	**variable** `=` **variable** `- 1;`

```
int x = 2;
x++;                    // x = x + 1;
                        // x now stores 3

double gpa = 2.5;
gpa--;                  // gpa = gpa - 1;
                        // gpa now stores 1.5
```

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

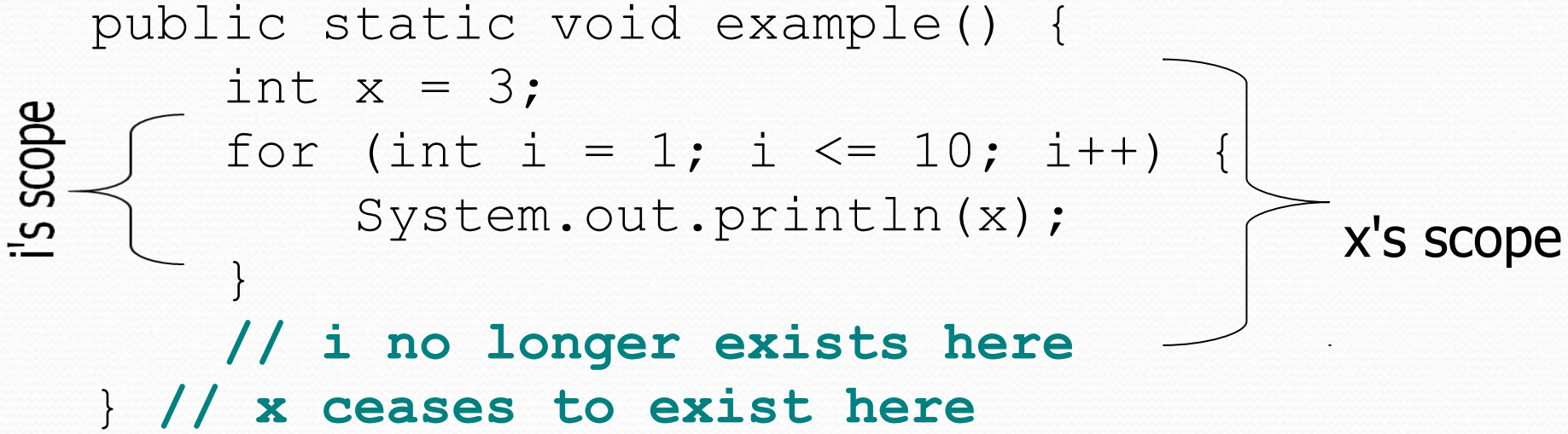
- **string concatenation**: Using + between a string and another value to make a longer string.

```
"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`

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.



Class constants

- **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;
```

Passing parameters

- Declaration:

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

- Call:

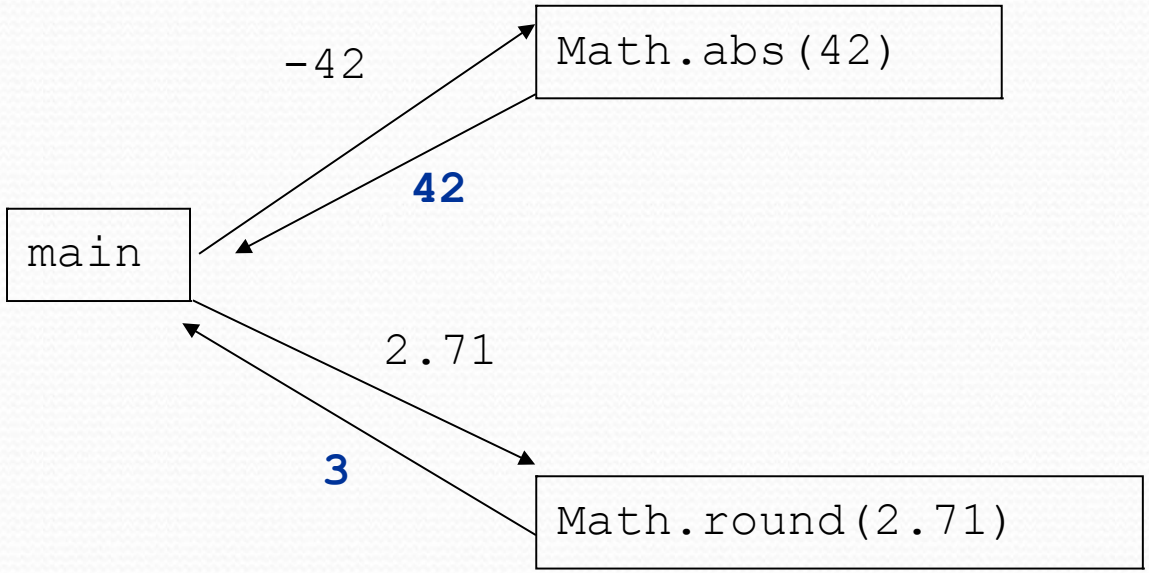
```
methodName (value, value, …, value);
```

- 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);
}
```

Return

- **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.



Java's `Math` class

Method name	Description
`Math.abs(`*value*`)`	absolute value
`Math.round(`*value*`)`	nearest whole number
`Math.ceil(`*value*`)`	rounds up
`Math.floor(`*value*`)`	rounds down
`Math.log10(`*value*`)`	logarithm, base 10
`Math.max(`*value1*`, `*value2*`)`	larger of two values
`Math.min(`*value1*`, `*value2*`)`	smaller of two values
`Math.pow(`*base*`, `*exp*`)`	*base* to the *exp* power
`Math.sqrt(`*value*`)`	square root

`Math.sin(`*value*`)`
`Math.cos(`*value*`)`
`Math.tan(`*value*`)`
	sine/cosine/tangent of an angle in radians

`Math.toDegrees(`*value*`)` `Math.toRadians(`*value*`)`	convert degrees to radians and back
`Math.random()`	random `double` between 0 and 1

Constant	Description
`Math.E`	2.7182818...
`Math.PI`	3.1415926...

Returning a value

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

- Example:

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

Strings

- **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	dy

  - 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

`toLowerCase()`	a new string with all lowercase letters
`toUpperCase()`	a new string with all uppercase letters

- These methods are called using the dot notation:

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

`String` test methods

Method	Description
`equals(`**str**`)`	whether two strings contain the same characters
`equalsIgnoreCase(`**str**`)`	whether two strings contain the same characters, ignoring upper vs. lower case
`startsWith(`**str**`)`	whether one contains other's characters at start
`endsWith(`**str**`)`	whether one contains other's characters at end
`contains(`**str**`)`	whether the given string is found within this one

```
String name = console.next();
if (name.startsWith("Dr.")) {
    System.out.println("Are you single?");
} else if (name.equalsIgnoreCase("LUMBERG")) {
    System.out.println("I need your TPS reports.");
}
```

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.

Type `char`

- `char` : A primitive type representing single characters.
  - Each character inside a `String` is stored as a `char` value.
  - Literal `char` values are surrounded with apostrophe (single-quote) marks, such as `'a'` or `'4'` or `'\n'` or `'\''`
  - It is legal to have variables, parameters, returns of type `char`

```
char letter = 'S';
System.out.println(letter);                    // S
```

- `char` values can be concatenated with strings.

```
char initial = 'P';
System.out.println(initial + " Diddy");   // P Diddy
```

`char` vs. `String`

- `"h"` is a `String`
  `'h'` is a `char` (the two behave differently)

- `String` is an object; it contains methods

```
String s = "h";
s = s.toUpperCase();          // 'H'
int len = s.length();         //  1
char first = s.charAt(0);     // 'H'
```

- `char` is primitive; you can't call methods on it

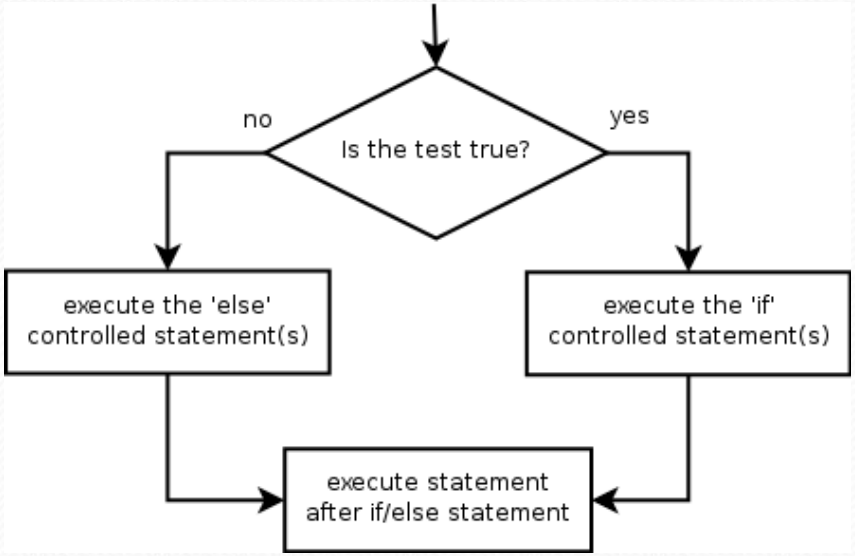
```
char c = 'h';
c = c.toUpperCase();    // ERROR: "cannot be dereferenced"
```

  - What is `s + 1`? What is `c + 1`?
  - What is `s + s`? What is `c + c`?

if/else

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.

- 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`

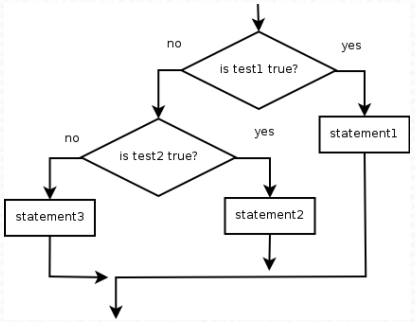
- **`boolean`**: A logical type whose values are `true` and `false`.
  - A **test** in an `if`, `for`, or `while` is a `boolean` expression.
  - You can create `boolean` variables, pass `boolean` parameters, return `boolean` values from methods, ...

```
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");
}
```

`if/else` Structures

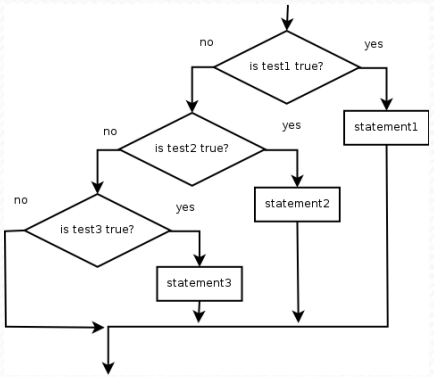
- Exactly 1 path: (mutually exclusive)

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



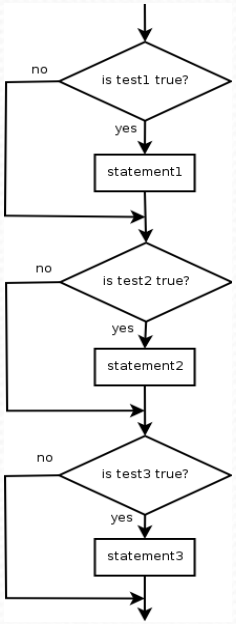
- 0 or 1 path:

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



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

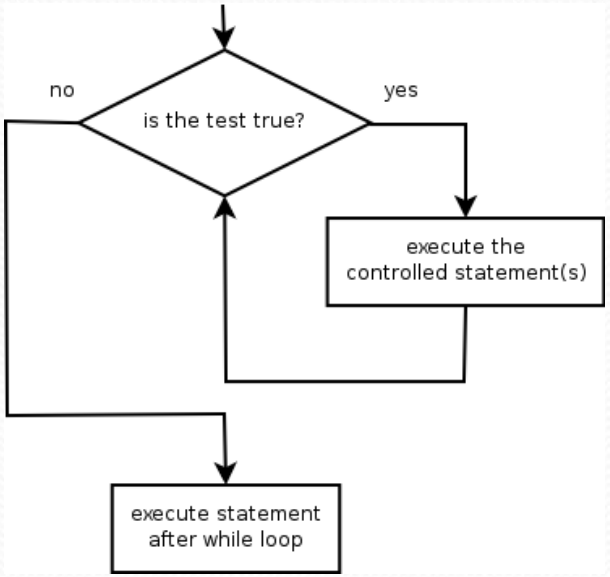
```
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
```



`while` loops

- **`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

- **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"));
```

The `Random` class

- A `Random` object generates pseudo-random* numbers.
  - Class `Random` is found in the `java.util` package.

```
import java.util.*;
```

Method name	Description
`nextInt()`	returns a random integer
`nextInt(`**max**`)`	returns a random integer in the range [0, *max*) in other words, 0 to *max*-1 inclusive
`nextDouble()`	returns a random real number in the range [0.0, 1.0)

  - Example:

```
Random rand = new Random();
int randomNumber = rand.nextInt(10);    // 0-9
```

`break`

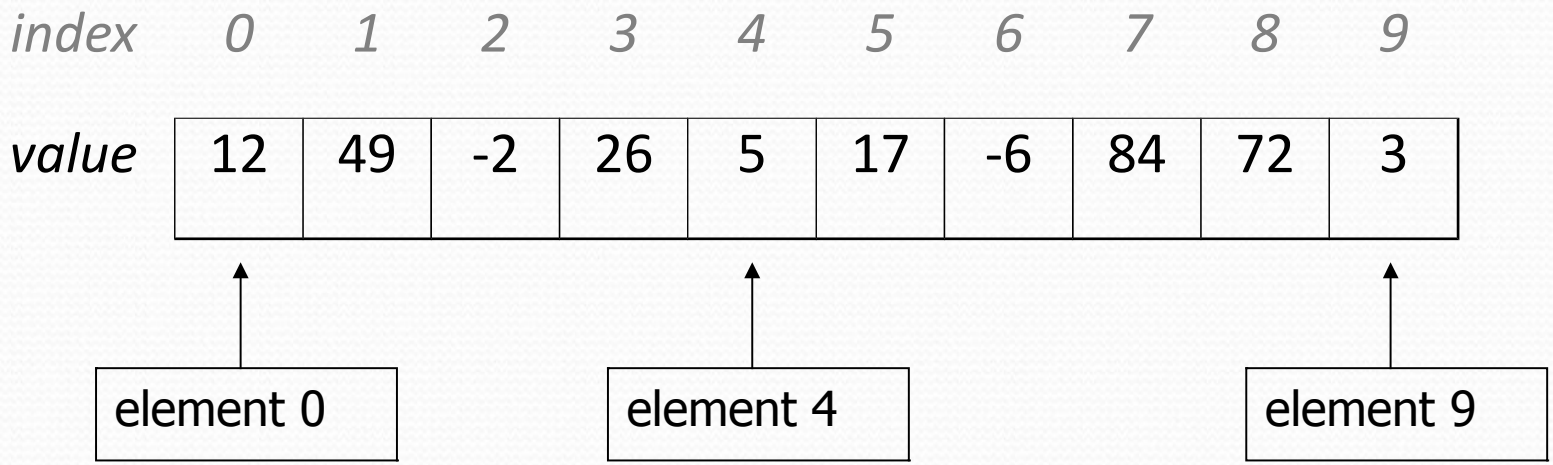
- **`break`** statement: Immediately exits a loop.
  - Can be used to write a loop whose test is in the middle.
  - Such loops are often called *"forever" loops* because their header's boolean test is often changed to a trivial `true`.

```
while (true) {
    statement(s);
    if (test) {
        break;
    }
    statement(s);
}
```

  - Some programmers consider `break` to be bad style.

Arrays

- **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];
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	0	0	0	0	0	0	0	0	0

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.");
}
```

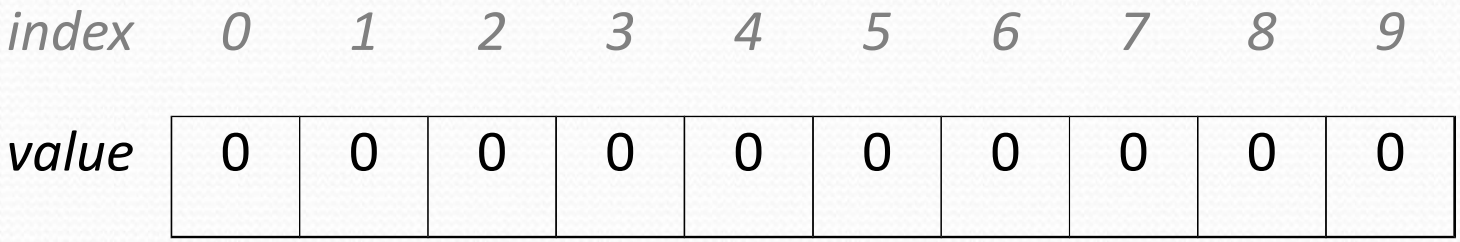
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	**27**	0	0	**-6**	0	0	0	0	0	0

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};
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
value	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 type methodName(type[] name) {
```

  - Example:

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

- Call:

```
methodName(arrayName);
```

  - Example:

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

Arrays as return

- Declaring:

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

  - Example:

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

- Calling:

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

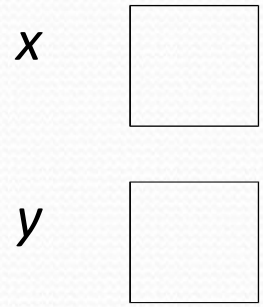
  - Example:

```
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.

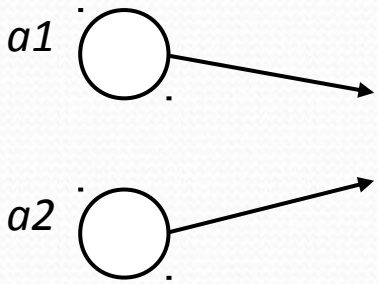
```
int x = 5;
int y = x;          // x = 5, y = 5
y = 17;             // x = 5, y = 17
x = 8;              // x = 8, y = 17
```



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
```

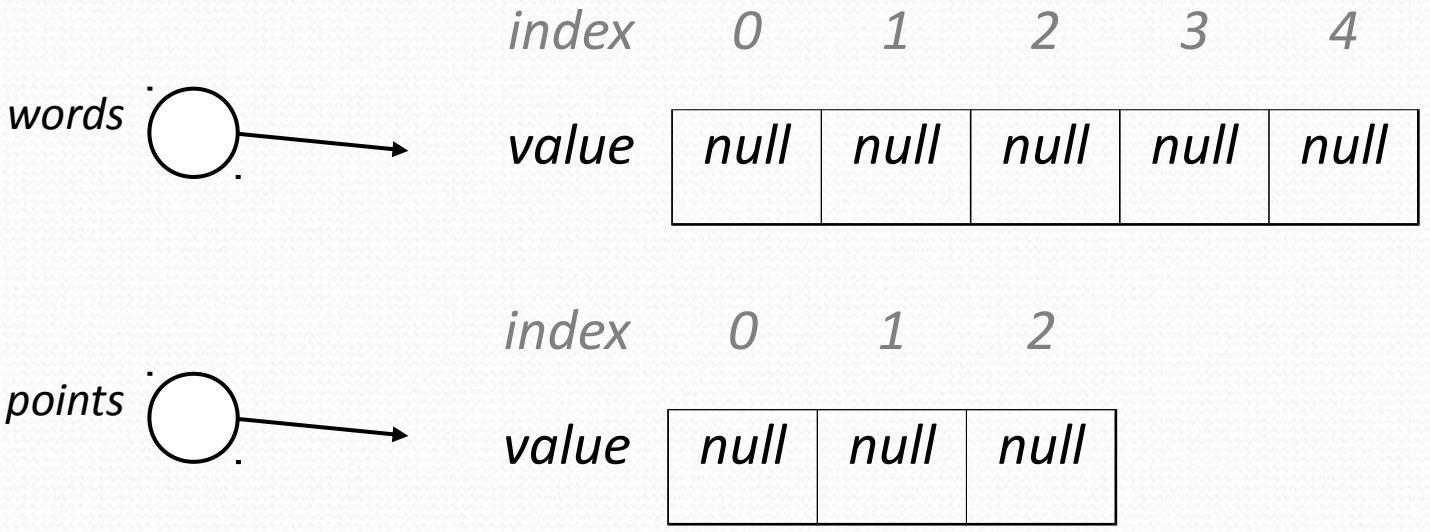


index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
value	**7**	5	2	12	14	14	9

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];
Point[] points = new Point[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

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

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

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

Fields

- **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.

- 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;
}
```

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

- **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

- 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

- **`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

- **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

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

- Example:

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

Overriding methods

- **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
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```

`super` keyword

- 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;
    }
    ...
}
```

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`.

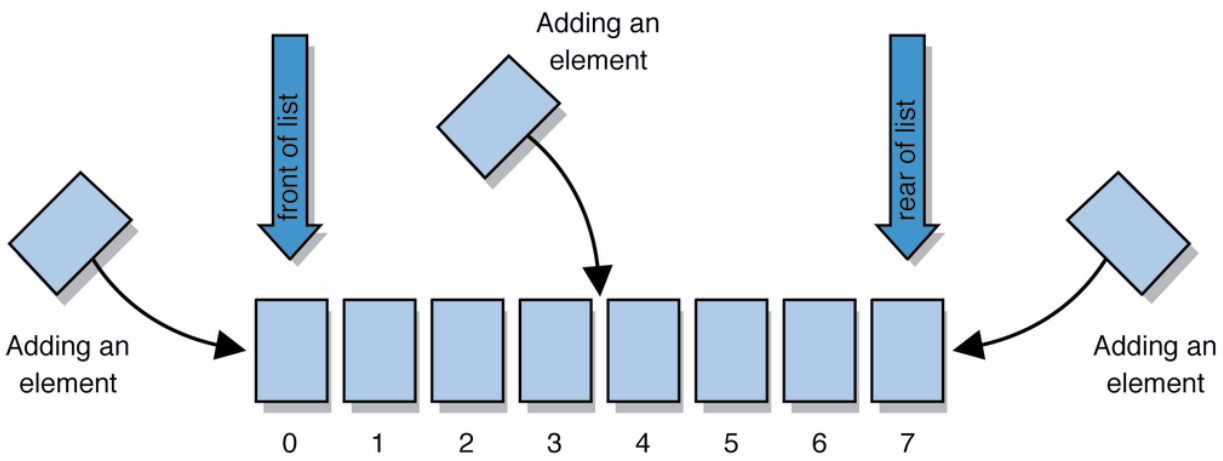
```
System.out.println(ed.getSalary());          // 55000.0
System.out.println(ed.getVacationForm());    // pink
```

Collections and lists

- **collection**: an object that stores data ("**elements**")

```
import java.util.*;   // to use Java's collections
```

- **list**: a collection of elements with 0-based **indexes**
  - elements can be added to the front, back, or elsewhere
  - a list has a **size** (number of elements that have been added)
  - in Java, a list can be represented as an **`ArrayList`** object



Idea of a list

- An `ArrayList` is like an array that resizes to fit its contents.

- When a list is created, it is initially empty.

```
[]
```

- You can add items to the list. (By default, adds at end of 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.
  - You can add, remove, get, set, ... any index at any time.

Type parameters (generics)

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

- When constructing an `ArrayList`, you must specify the type of its elements in < >
  - This is called a *type parameter* ; `ArrayList` is a *generic* class.
  - Allows the `ArrayList` class to store lists of different types.

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

`ArrayList` methods

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`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(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

ArrayList vs. array

```
String[] names = new String[5];              // construct
names[0] = "Jessica";                        // store
String s = names[0];                         // retrieve
for (int i = 0; i < names.length; i++) {
    if (names[i].startsWith("B")) { ... }
}                                            // iterate
```

```
ArrayList<String> list = new ArrayList<String>();
list.add("Jessica");                         // store
String s = list.get(0);                      // retrieve
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}                                            // iterate
```

`ArrayList` as param/return

```
public void name(ArrayList<Type> name) {    // param
public ArrayList<Type> name(params)    // return
```

- Example:

```
// Returns count of plural words in the given list.
public int countPlural(ArrayList<String> list) {
    int count = 0;
    for (int i = 0; i < list.size(); i++) {
        String str = list.get(i);
        if (str.endsWith("s")) {
            count++;
        }
    }
    return count;
}
```

Throwing exceptions

```
throw new ExceptionType();
throw new ExceptionType("message");
```

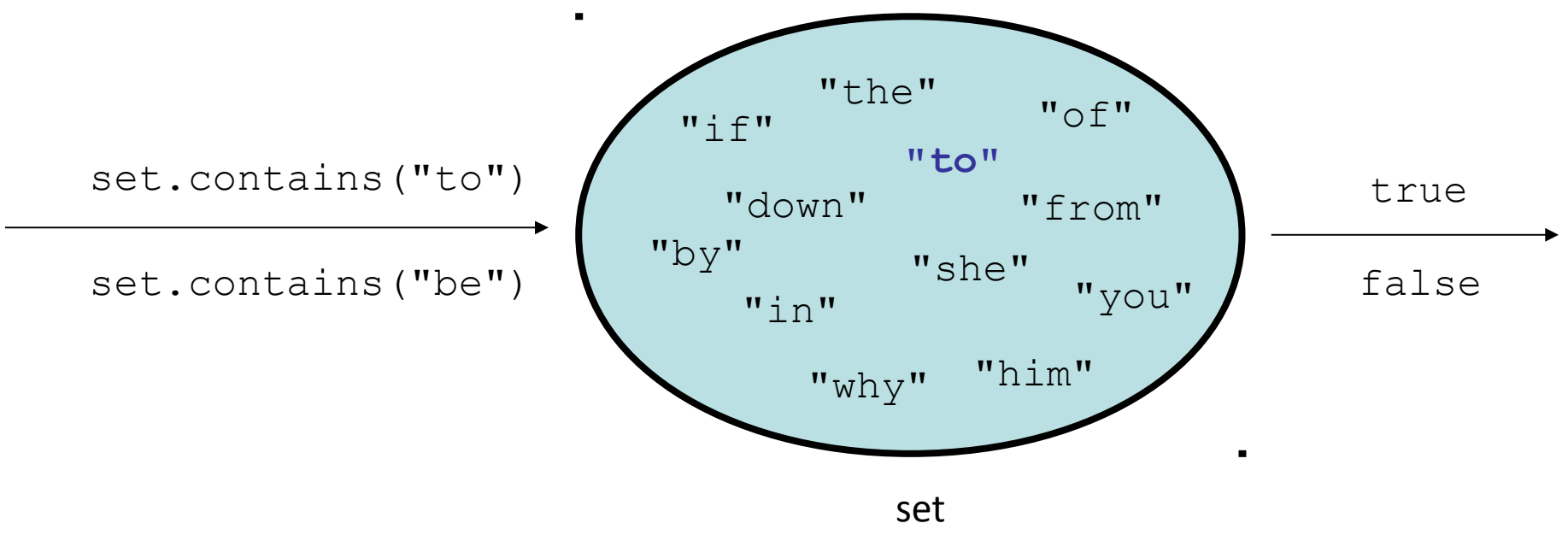
- Generates an exception that will crash the program, unless it has code to handle ("catch") the exception.

- Common exception types:
  - ArithmeticException, ArrayIndexOutOfBoundsException, FileNotFoundException, IllegalArgumentException, IllegalStateException, IOException, NoSuchElementException, NullPointerException, RuntimeException, UnsupportedOperationException

- Why would anyone ever *want* a program to crash?

Sets

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



`Set` implementation

- in Java, sets are represented by `Set` type in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
  - `HashSet`: implemented using a "hash table" array;
    very fast: **O(1)** for all operations
    elements are stored in unpredictable order
  - `TreeSet`: implemented using a "binary search tree";
    pretty fast: **O(log N)** for all operations
    elements are stored in sorted order
  - `LinkedHashSet`: **O(1)** but stores in order of insertion;
    slightly slower than `HashSet` because of extra info stored

`Set` methods

```
List<String> list = new ArrayList<String>();
…
Set<Integer> set = new TreeSet<Integer>();      // empty
Set<String> set2 = new HashSet<String>(list);
```

- can construct an empty set, or one based on a given collection

`add(`**value**`)`	adds the given value to the set
`contains(`**value**`)`	returns `true` if the given value is found in this set
`remove(`**value**`)`	removes the given value from the set
`clear()`	removes all elements of the set
`size()`	returns the number of elements in list
`isEmpty()`	returns `true` if the set's size is 0
`toString()`	returns a string such as `"[3, 42, -7, 15]"`

The "for each" loop

```
for (type name : collection) {
    statements;
}
```

- Provides a clean syntax for looping over the elements of a `Set`, `List`, array, or other collection

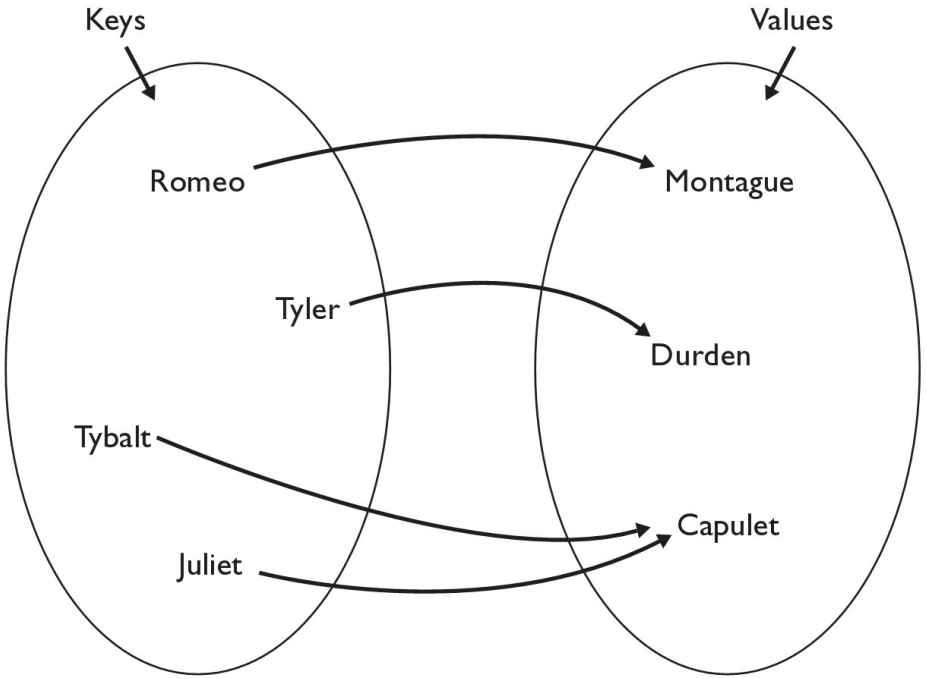
```
Set<Double> grades = new HashSet<Double>();
…

for (double grade : grades) {
    System.out.println("Student's grade: " + grade);
}
```

  - needed because sets have no indexes; can't `get` element `i`

Maps

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"

- basic map operations:
  - **put**(*key*, *value*): Adds a mapping from a key to a value.
  - **get**(*key*): Retrieves the value mapped to the key.
  - **remove**(*key*): Removes the given key and its mapped value.



`myMap.get("Juliet")` returns `"Capulet"`

Map implementation

- in Java, maps are represented by `Map` type in `java.util`
- `Map` is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap`: implemented using an array called a "hash table"; extremely fast: **O(1)** ; keys are stored in unpredictable order
  - `TreeMap`: implemented as a linked "binary tree" structure; very fast: **O(log N)** ; keys are stored in sorted order
  - `LinkedHashMap`: O(1) ; keys are stored in order of insertion
- A map requires 2 type params: one for keys, one for values.

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```

Map methods

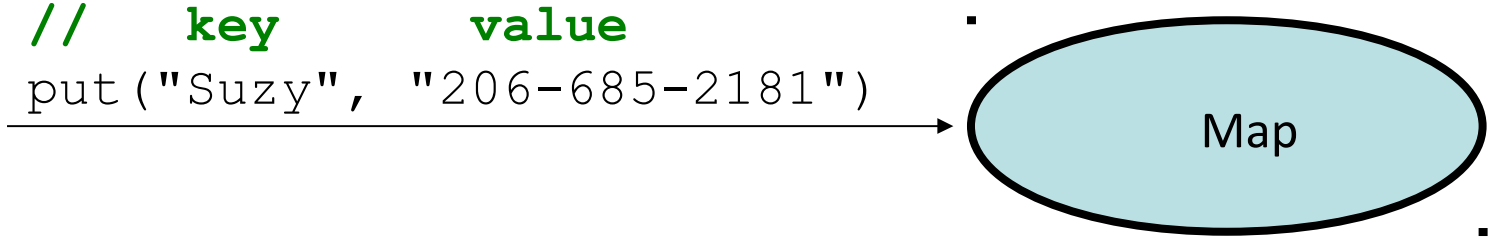
`put(`**key**`, `**value**`)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**key**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**key**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**key**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`

`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

Using maps

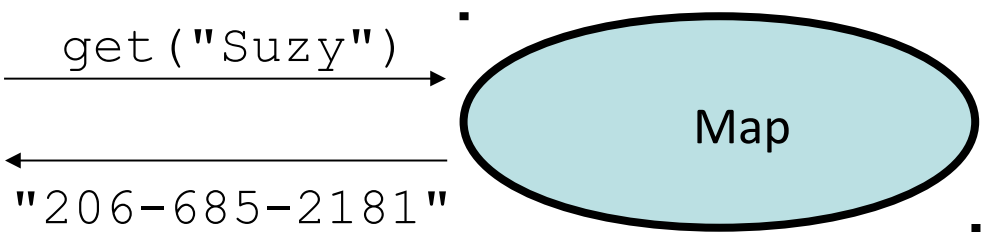
- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

```
//   key       value
put("Suzy", "206-685-2181")
```



  - Later, we can supply only the key and get back the related value:

    Allows us to ask: *What is Suzy's phone number?*



keySet and values

- `keySet` method returns a `Set` of all keys in the map
  - can loop over the keys in a foreach loop
  - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);  // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) {              // Geneva -> 2
    int age = ages.get(name);                    // Marty -> 19
    System.out.println(name + " -> " + age);     // Vicki -> 57
}
```

- `values` method returns a collection of all values in the map
  - can loop over the values in a foreach loop
  - no easy way to get from a value to its associated key(s)

The `compareTo` method

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```

- A call of **A.**`compareTo(`**B**`)` will return:

a value < 0 if **A** comes "before" **B** in the ordering,

a value > 0 if **A** comes "after" **B** in the ordering,

or 0 if **A** and **B** are considered "equal" in the ordering.

Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {   // true
    ...
}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

`compareTo` and collections

- You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

- Java's `TreeSet`/`Map` use `compareTo` internally for ordering.

```
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Ordering our own types

- We cannot binary search or make a `TreeSet`/`Map` of arbitrary types, because Java doesn't know how to order the elements.
  - The program compiles but crashes when we run it.

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));
tags.add(new HtmlTag("b", false));
…
```

```
Exception in thread "main" java.lang.ClassCastException
        at java.util.TreeSet.add(TreeSet.java:238)
```

Comparable

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.

- A call to your `compareTo` method should return:

  a value < 0 if `this` object comes "before" the `other` object,

  a value > 0 if `this` object comes "after" the `other` object,

  or0 if `this` object is considered "equal" to the `other`.

- If you want multiple orderings, use a `Comparator` instead (see Ch. 13.1)

Comparable example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    …

    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;     // same x, larger y
        } else {
            return 0;     // same x and same y
        }
    }
}
```

`Collections` class

Method name	Description
`binarySearch(`**list**`, `**value**`)`	returns the index of the given value in a sorted list (< 0 if not found)
`copy(`**listTo**`, `**listFrom**`)`	copies **listFrom**'s elements to **listTo**
`emptyList(), emptyMap(), emptySet()`	returns a read-only collection of the given type that has no elements
`fill(`**list**`, `**value**`)`	sets every element in the list to have the given value
`max(`**collection**`), min(`**collection**`)`	returns largest/smallest element
`replaceAll(`**list**`, `**old**`, `**new**`)`	replaces an element value with another
`reverse(`**list**`)`	reverses the order of a list's elements
`shuffle(`**list**`)`	arranges elements into a random order
`sort(`**list**`)`	arranges elements into ascending order

Sorting methods in Java

- The `Arrays` and `Collections` classes in `java.util` have a static method `sort` that sorts the elements of an array/list

```
String[] words = {"foo", "bar", "baz", "ball"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
// [ball, bar, baz, foo]

List<String> words2 = new ArrayList<String>();
for (String word : words) {
    words2.add(word);
}
Collections.sort(words2);
System.out.println(words2);
// [ball, bar, baz, foo]
```

Recall: Inheritance

- **inheritance**: Forming new classes based on existing ones.
  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
    - gets a copy of every field and method from superclass

- **override**: To replace a superclass's method by writing a new version of that method in a subclass.

```
public class Lawyer extends Employee {
    // overrides getSalary in Employee; a raise!
    public double getSalary() {
        return 55000.00;
    }
}
```

The super keyword

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

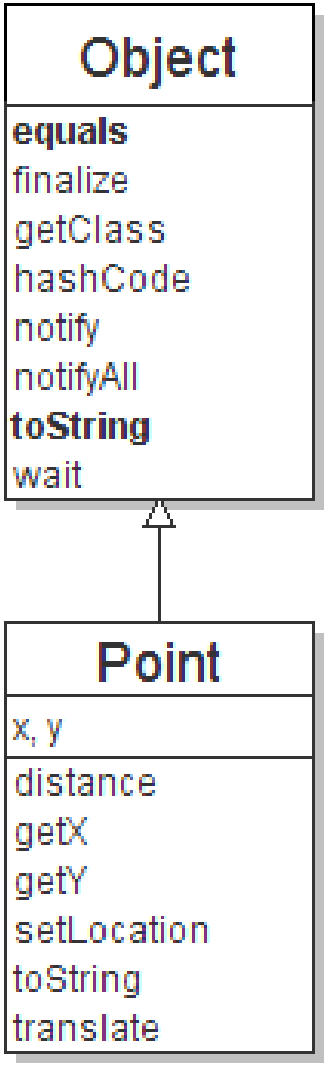
- Subclasses can call overridden methods/constructors with `super`

```
public class Lawyer extends Employee {
    private boolean passedBarExam;

    public Lawyer(int vacationDays, boolean bar) {
        super(vacationDays * 2);
        this.passedBarExam = bar;
    }

    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;   // $5K raise
    }
    ...
}
```

The class Object

- The class `Object` forms the root of the overall inheritance tree of all Java classes.
  - Every class is implicitly a subclass of `Object`

- The `Object` class defines several methods that become part of every class you write. For example:
  - `public String toString()`
    Returns a text representation of the object, usually so that it can be printed.



Object methods

method	description
`protected Object clone()`	creates a copy of the object
`public boolean equals(Object o)`	returns whether two objects have the same state
`protected void finalize()`	used for garbage collection
`public Class<?> getClass()`	info about the object's type
`public int hashCode()`	a code suitable for putting this object into a hash collection
`public String toString()`	text representation of object

`public void notify()`
`public void notifyAll()`
`public void wait()`
`public void wait(...)`
	methods related to concurrency and locking (seen later)

- What does this list of methods tell you about Java's design?

Using the Object class

- You can store any object in a variable of type `Object`.

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";
```

- You can write methods that accept an `Object` parameter.

```
public void checkNotNull(Object o) {
    if (o != null) {
        throw new IllegalArgumentException();
    }
```

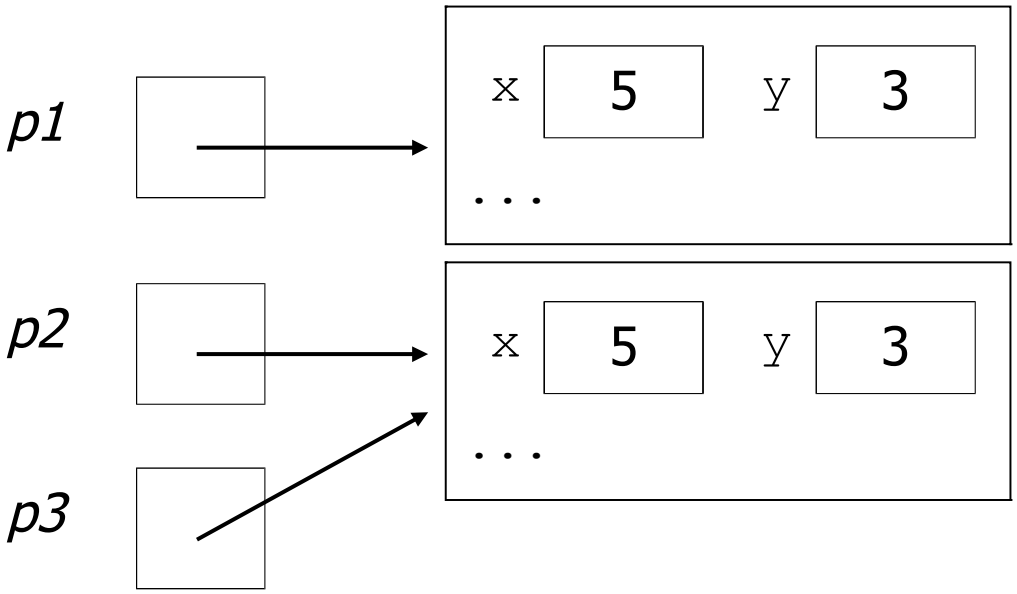
- You can make arrays or collections of `Objects`.

```
Object[] a = new Object[5];
a[0] = "hello";
a[1] = new Random();
List<Object> list = new ArrayList<Object>();
```

Recall: comparing objects

- The == operator does not work well with objects.
  - It compares references, not objects' state.
  - It produces `true` only when you compare an object to itself.

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
Point p3 = p2;

// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true

// p1.equals(p2)?
// p2.equals(p3)?
```



Default equals method

- The `Object` class's `equals` implementation is very simple:

```
public class Object {
    ...
    public boolean equals(Object o) {
        return this == o;
    }
}
```

- However:
  - When we have used equals with various objects, it didn't behave like `==`. Why not? `if (str1.equals(str2)) { ...`
  - The Java API documentation for equals is elaborate. Why?

Implementing equals

```
public boolean equals(Object name) {
    statement(s) that return a boolean value ;
}
```

- The parameter to `equals` must be of type `Object`.
- Having an `Object` parameter means *any* object can be passed.
  - If we don't know what type it is, how can we compare it?

Casting references

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";

((Point) o1).translate(6, 2);       // ok
int len = ((String) o2).length();   // ok
Point p = (Point) o1;
int x = p.getX();                   // ok
```

- Casting references is different than casting primitives.
  - Really casting an `Object` reference into a `Point` reference.
  - Doesn't actually change the object that is referred to.
  - Tells the compiler to *assume* that `o1` refers to a `Point` object.

The instanceof keyword

```
if (variable instanceof type) {
    statement(s);
}
```

- Asks if a variable refers to an object of a given type.
  - Used as a `boolean` test.

```
String s = "hello";
Point p = new Point();
```

expression	result
`s instanceof Point`	`false`
`s instanceof String`	`true`
`p instanceof Point`	`true`
`p instanceof String`	`false`
`p instanceof Object`	`true`
`s instanceof Object`	`true`
`null instanceof String`	`false`
`null instanceof Object`	`false`

equals method for Points

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point.
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; cannot be equal
        return false;
    }
}
```

More about equals

- Equality is expected to be reflexive, symmetric, and transitive:

  `a.equals(a)` is `true` for every object `a`
  `a.equals(b)` ↔ `b.equals(a)`
  `(a.equals(b) && b.equals(c))` ↔ `a.equals(c)`

- No non-`null` object is equal to `null`:

  `a.equals(null)` is `false` for every object `a`

- Two sets are equal if they contain the same elements:

```
Set<String> set1 = new HashSet<String>();
Set<String> set2 = new TreeSet<String>();
for (String s : "hi how are you".split(" ")) {
    set1.add(s);    set2.add(s);
}
System.out.println(set1.equals(set2));   // true
```

The hashCode method

`public int hashCode()`

Returns an integer hash code for this object, indicating its preferred to place it in a hash table / hash set.

- Allows us to store non-`int` values in a hash set/map:

```
public static int hashFunction(Object o) {
    return Math.abs(o.hashCode()) % elements.length;
}
```

- How is `hashCode` implemented?
  - Depends on the type of object and its state.
    - Example: a String's `hashCode` adds the ASCII values of its letters.
  - You can write your own `hashCode` methods in classes you write.
    - All classes come with a default version based on memory address.

Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

- A variable or parameter of type *T* can refer to any subclass of *T*.

```
Employee ed = new Lawyer();
Object otto = new Secretary();
```

  - When a method is called on `ed`, it behaves as a `Lawyer`.
    - You can call any `Employee` methods on `ed`.
      You can call any `Object` methods on `otto`.
      - You can *not* call any `Lawyer`-only methods on `ed` (e.g. `sue`).
        You can *not* call any `Employee` methods on `otto` (e.g. `getHours`).

Polymorphism examples

- You can use the object's extra functionality by casting.

```
Employee ed = new Lawyer();
ed.getVacationDays();                    // ok
ed.sue();                                // compiler error
((Lawyer) ed).sue();                     // ok
```

- You can't cast an object into something that it is not.

```
Object otto = new Secretary();
System.out.println(otto.toString());     // ok
otto.getVacationDays();                  // compiler error
((Employee) otto).getVacationDays();     // ok
((Lawyer) otto).sue();                   // runtime error
```