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 declaredinitialized in one statement.

- Syntax:
  
  type name = value;

  - double myGPA = 3.95;

  - int x = (11 % 3) + 12;
Java's primitive types

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integers</td>
<td>42, -3, 0, 926394</td>
</tr>
<tr>
<td>double</td>
<td>real numbers</td>
<td>3.1, -0.25, 9.4e3</td>
</tr>
</tbody>
</table>
| char      | single text characters    | 'a', 'X', '?', '
' |
| 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:

```java
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

<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Equivalent longer version</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable++</td>
<td>variable = variable + 1;</td>
</tr>
<tr>
<td>variable--</td>
<td>variable = variable - 1;</td>
</tr>
</tbody>
</table>

```java
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.

```java
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

• **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:

  ```java
  public static final type name = value;
  ```
  - name is usually in ALL_UPPER_CASE

  Examples:
  ```java
  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:

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

- Call:

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

- Example:

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

  ```java
  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.

```
// Example of return values
int main() {
    int x = Math.abs(42);  // x = 42
    return x;
}
```

```
// Example of return values
double y = Math.round(2.71);  // y = 3.0
return y;
```
# Java's Math class

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Math.abs(value)</code></td>
<td>absolute value</td>
</tr>
<tr>
<td><code>Math.round(value)</code></td>
<td>nearest whole number</td>
</tr>
<tr>
<td><code>Math.ceil(value)</code></td>
<td>rounds up</td>
</tr>
<tr>
<td><code>Math.floor(value)</code></td>
<td>rounds down</td>
</tr>
<tr>
<td><code>Math.log10(value)</code></td>
<td>logarithm, base 10</td>
</tr>
<tr>
<td><code>Math.max(value1, value2)</code></td>
<td>larger of two values</td>
</tr>
<tr>
<td><code>Math.min(value1, value2)</code></td>
<td>smaller of two values</td>
</tr>
<tr>
<td><code>Math.pow(base, exp)</code></td>
<td><code>base</code> to the <code>exp</code> power</td>
</tr>
<tr>
<td><code>Math.sqrt(value)</code></td>
<td>square root</td>
</tr>
<tr>
<td><code>Math.sin(value)</code></td>
<td>sine/cosine/tangent of an angle in radians</td>
</tr>
<tr>
<td><code>Math.cos(value)</code></td>
<td></td>
</tr>
<tr>
<td><code>Math.tan(value)</code></td>
<td></td>
</tr>
<tr>
<td><code>Math.toDegrees(value)</code></td>
<td>convert degrees to radians</td>
</tr>
<tr>
<td><code>Math.toRadians(value)</code></td>
<td>radians and back</td>
</tr>
<tr>
<td><code>Math.random()</code></td>
<td>random double between 0 and 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.E</td>
<td>2.7182818...</td>
</tr>
<tr>
<td>Math.PI</td>
<td>3.1415926...</td>
</tr>
</tbody>
</table>
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.

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

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

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

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>P</td>
<td>.</td>
<td>D</td>
<td>i</td>
<td>d</td>
<td>d</td>
<td>dy</td>
<td></td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>indexOf(str)</code></td>
<td>index where the start of the given string appears in this string (-1 if it is not there)</td>
</tr>
<tr>
<td><code>length()</code></td>
<td>number of characters in this string</td>
</tr>
<tr>
<td><code>substring(index1, index2)</code> or <code>substring(index1)</code></td>
<td>the characters in this string from <code>index1</code> (inclusive) to <code>index2</code> (exclusive); if <code>index2</code> omitted, grabs till end of string</td>
</tr>
<tr>
<td><code>toLowerCase()</code></td>
<td>a new string with all lowercase letters</td>
</tr>
<tr>
<td><code>toUpperCase()</code></td>
<td>a new string with all uppercase letters</td>
</tr>
</tbody>
</table>

- These methods are called using the dot notation:

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals(str)</td>
<td>whether two strings contain the same characters</td>
</tr>
<tr>
<td>equalsIgnoreCase(str)</td>
<td>whether two strings contain the same characters, ignoring upper vs. lower case</td>
</tr>
<tr>
<td>startsWith(str)</td>
<td>whether one contains other's characters at start</td>
</tr>
<tr>
<td>endsWith(str)</td>
<td>whether one contains other's characters at end</td>
</tr>
<tr>
<td>contains(str)</td>
<td>whether the given string is found within this one</td>
</tr>
</tbody>
</table>

```java
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.

```java
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if ("Barney").equals(name) { // Technically this is a method that returns a value of type boolean, the type used in logical tests.
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```
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 '
' or '
'
  - It is legal to have variables, parameters, returns of type `char`

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

- `char` values can be concatenated with strings.

```java
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

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

- Example:
  ```java
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.
  
  ```java
  for (int i = 1; i <= 10; i++) {
      ...
  }
  if (i <= 10) {
      ...
  }
  ```
  
  - These are boolean expressions.

- Tests use relational operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>equals</td>
<td><code>1 + 1 == 2</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>!=</code></td>
<td>does not equal</td>
<td><code>3.2 != 2.5</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
<td><code>10 &lt; 5</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
<td><code>10 &gt; 5</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to</td>
<td><code>126 &lt;= 100</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to</td>
<td><code>5.0 &gt;= 5.0</code></td>
<td><code>true</code></td>
</tr>
</tbody>
</table>
Logical operators: &&, ||, !

- Conditions can be combined using *logical operators*:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>(2 == 3) &amp;&amp; (-1 &lt; 5)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>!(2 == 3)</td>
<td>true</td>
</tr>
</tbody>
</table>

- "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     |

<table>
<thead>
<tr>
<th>$p$</th>
<th>! $p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
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, ...

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

- **0 or 1 path:**
  ```java
  if (test) {
    statement(s);
  } else if (test) {
    statement(s);
  } else if (test) {
    statement(s);
  }
  ```

- **0, 1, or many paths: (independent tests, not exclusive)**
  ```java
  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.

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

- **Example:**

  ```java
  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.

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

- **Example:**

  ```java
  // 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.

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

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextInt()</td>
<td>returns a random integer</td>
</tr>
<tr>
<td>nextInt(max)</td>
<td>returns a random integer in the range [0, max) in other words, 0 to max-1 inclusive</td>
</tr>
<tr>
<td>nextDouble()</td>
<td>returns a random real number in the range [0.0, 1.0)</td>
</tr>
</tbody>
</table>

- Example:

```java
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`.

```java
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];

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Accessing elements

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

- Example:

```numbers``` = 27;
n```numbers``` = -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:

  ```java
  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
  ```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The length field

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

```java
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};

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>12</td>
<td>49</td>
<td>-2</td>
<td>26</td>
<td>5</td>
<td>17</td>
<td>-6</td>
</tr>
</tbody>
</table>

• 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:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(array, value)</td>
<td>returns the index of the given value in a sorted array (&lt; 0 if not found)</td>
</tr>
<tr>
<td>equals(array1, array2)</td>
<td>returns true if the two arrays contain the same elements in the same order</td>
</tr>
<tr>
<td>fill(array, value)</td>
<td>sets every element in the array to have the given value</td>
</tr>
<tr>
<td>sort(array)</td>
<td>arranges the elements in the array into ascending order</td>
</tr>
<tr>
<td>toString(array)</td>
<td>returns a string representing the array, such as &quot;[10, 30, 17]&quot;</td>
</tr>
</tbody>
</table>
Arrays as parameters

• Declaration:
  ```java
  public type methodName(type[] name) {
  ```

  • Example:
  ```java
  public double average(int[] numbers) {
      ...
  }
  ```

• Call:
  ```java
  methodName(arrayName);
  ```

  • Example:
  ```java
  int[] scores = {13, 17, 12, 15, 11};
  double avg = average(scores);
  ```
Arrays as return

• Declaring:
  ```java
  public type[] methodName (parameters) {
  ```
  
  • Example:
  ```java
  public int[] countDigits(int n) {
    int[] counts = new int[10];
    ...  
    return counts;
  }
  ```

• Calling:
  ```java
  type[] name = methodName (parameters);  
  ```
  
  • Example:
  ```java
  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.

```java
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.

```java
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
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

```
 a1
-----
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>9</td>
</tr>
</tbody>
</table>
 a2
```
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`.

```java
String[] words = new String[5];
Point[] points = new Point[3];
```

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>4</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>null</td>
</tr>
</tbody>
</table>
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.

```java
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:

  ```java
  private type name;
  ```

- Example:

  ```java
  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.

  ```java
  public type name(parameters) {
    statements;
  }
  
  Example:
  public void shout() {
    System.out.println("HELLO THERE!");
  }
  ```
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 \texttt{p1.draw(g)};
    the object referred to by \texttt{p1} is the implicit parameter.
  
  – During the call \texttt{p2.draw(g)};
    the object referred to by \texttt{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.
    • \texttt{draw} can refer to the \texttt{x} and \texttt{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.

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

  - Example:
    ```java
    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
  ```java
  public String toString() {
    code that returns a suitable String;
  }
  ```
- Example:
  ```java
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
  ```
- called when an object is printed/concatenated to a String:
  ```java
  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:

```java
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

```java
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.

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

- Subclasses can call overridden methods with `super`

  ```java
  super.method(parameters)
  ```

- Example:

  ```java
  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 `$T$`.
  ```java
  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`.
  ```java
  System.out.println(ed.getSalary()); // 55000.0
  System.out.println(ed.getVacationForm()); // pink
  ```
Collections and lists

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

```java
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)

```java
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.

```java
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(value)</code></td>
<td>appends value at end of list</td>
</tr>
<tr>
<td><code>add(index, value)</code></td>
<td>inserts given value just before the given index, shifting subsequent values to the right</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all elements of the list</td>
</tr>
<tr>
<td><code>indexOf(value)</code></td>
<td>returns first index where given value is found in list (-1 if not found)</td>
</tr>
<tr>
<td><code>get(index)</code></td>
<td>returns the value at given index</td>
</tr>
<tr>
<td><code>remove(index)</code></td>
<td>removes/returns value at given index, shifting subsequent values to the left</td>
</tr>
<tr>
<td><code>set(index, value)</code></td>
<td>replaces value at given index with given value</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in list</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>returns a string representation of the list such as &quot;[3, 42, -7, 15]&quot;</td>
</tr>
</tbody>
</table>
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**

```java
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

```java
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.

```java
set.contains("to") // true
set.contains("be") // false
```
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

```java
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(value)</code></td>
<td>adds the given value to the set</td>
</tr>
<tr>
<td><code>contains(value)</code></td>
<td>returns <code>true</code> if the given value is found in this set</td>
</tr>
<tr>
<td><code>remove(value)</code></td>
<td>removes the given value from the set</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all elements of the set</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in list</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns <code>true</code> if the set's size is 0</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>returns a string such as &quot;[3, 42, -7, 15]&quot;</td>
</tr>
</tbody>
</table>
The "for each" loop

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

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

```java
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.

```java
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.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(key, value)</code></td>
<td>adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one</td>
</tr>
<tr>
<td><code>get(key)</code></td>
<td>returns the value mapped to the given key (null if not found)</td>
</tr>
<tr>
<td><code>containsKey(key)</code></td>
<td>returns true if the map contains a mapping for the given key</td>
</tr>
<tr>
<td><code>remove(key)</code></td>
<td>removes any existing mapping for the given key</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all key/value pairs from the map</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of key/value pairs in the map</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns true if the map's size is 0</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>returns a string such as &quot;{a=90, d=60, c=70}&quot;</td>
</tr>
<tr>
<td><code>keySet()</code></td>
<td>returns a set of all keys in the map</td>
</tr>
<tr>
<td><code>values()</code></td>
<td>returns a collection of all values in the map</td>
</tr>
<tr>
<td><code>putAll(map)</code></td>
<td>adds all key/value pairs from the given map to this map</td>
</tr>
<tr>
<td><code>equals(map)</code></td>
<td>returns true if given map has the same mappings as this one</td>
</tr>
</tbody>
</table>
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

```java
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()) {
    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:
    ```java
    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.

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

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &lt; b) { ...</td>
<td>if (a.compareTo(b) &lt; 0) { ...</td>
</tr>
<tr>
<td>if (a &lt;= b) { ...</td>
<td>if (a.compareTo(b) &lt;= 0) { ...</td>
</tr>
<tr>
<td>if (a == b) { ...</td>
<td>if (a.compareTo(b) == 0) { ...</td>
</tr>
<tr>
<td>if (a != b) { ...</td>
<td>if (a.compareTo(b) != 0) { ...</td>
</tr>
<tr>
<td>if (a &gt;= b) { ...</td>
<td>if (a.compareTo(b) &gt;= 0) { ...</td>
</tr>
<tr>
<td>if (a &gt; b) { ...</td>
<td>if (a.compareTo(b) &gt; 0) { ...</td>
</tr>
</tbody>
</table>
compareTo and collections

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

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

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

```java
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(set);
// [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.

```java
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,
  or 0 if this object is considered "equal" to the other.

• If you want multiple orderings, use a Comparator instead (see Ch. 13.1)
Comparable example

```java
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

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(list, value)</td>
<td>returns the index of the given value in a sorted list (&lt; 0 if not found)</td>
</tr>
<tr>
<td>copy(listTo, listFrom)</td>
<td>copies listFrom's elements to listTo</td>
</tr>
<tr>
<td>emptyList(), emptyMap(), emptySet()</td>
<td>returns a read-only collection of the given type that has no elements</td>
</tr>
<tr>
<td>fill(list, value)</td>
<td>sets every element in the list to have the given value</td>
</tr>
<tr>
<td>max(collection), min(collection)</td>
<td>returns largest/smallest element</td>
</tr>
<tr>
<td>replaceAll(list, old, new)</td>
<td>replaces an element value with another</td>
</tr>
<tr>
<td>reverse(list)</td>
<td>reverses the order of a list's elements</td>
</tr>
<tr>
<td>shuffle(list)</td>
<td>arranges elements into a random order</td>
</tr>
<tr>
<td>sort(list)</td>
<td>arranges elements into ascending order</td>
</tr>
</tbody>
</table>
The Arrays and Collections classes in java.util have a static method sort that sorts the elements of an array/list.

```java
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.

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

super.\texttt{method}(\texttt{parameters})
\texttt{super}(\texttt{parameters});

\begin{itemize}
  \item Subclasses can call overridden methods/constructors with \texttt{super}
\end{itemize}

\begin{verbatim}
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
  }

  ...
}
\end{verbatim}
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

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected Object <strong>clone()</strong></td>
<td>creates a copy of the object</td>
</tr>
<tr>
<td>public boolean <strong>equals</strong>(Object o)</td>
<td>returns whether two objects have the same state</td>
</tr>
<tr>
<td>protected void <strong>finalize()</strong></td>
<td>used for garbage collection</td>
</tr>
<tr>
<td>public Class&lt;?]** getClass()**</td>
<td>info about the object's type</td>
</tr>
<tr>
<td>public int <strong>hashCode()</strong></td>
<td>a code suitable for putting this object into a hash collection</td>
</tr>
<tr>
<td>public String <strong>toString()</strong></td>
<td>text representation of object</td>
</tr>
<tr>
<td>public void <strong>notify()</strong></td>
<td>methods related to concurrency and locking (seen later)</td>
</tr>
<tr>
<td>public void <strong>notifyAll()</strong></td>
<td></td>
</tr>
<tr>
<td>public void <strong>wait()</strong></td>
<td></td>
</tr>
<tr>
<td>public void <strong>wait(...)</strong></td>
<td></td>
</tr>
</tbody>
</table>

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

- You can write methods that accept an `Object` parameter.
  ```java
  public void checkNotNull(Object o) {
      if (o != null) {
          throw new IllegalArgumentException();
      }
  }
  ```

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

```java
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:

```java
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?
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();

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>s instanceof Point</td>
<td>false</td>
</tr>
<tr>
<td>s instanceof String</td>
<td>true</td>
</tr>
<tr>
<td>p instanceof Point</td>
<td>true</td>
</tr>
<tr>
<td>p instanceof String</td>
<td>false</td>
</tr>
<tr>
<td>p instanceof Object</td>
<td>true</td>
</tr>
<tr>
<td>s instanceof Object</td>
<td>true</td>
</tr>
<tr>
<td>null instanceof String</td>
<td>false</td>
</tr>
<tr>
<td>null instanceof Object</td>
<td>false</td>
</tr>
</tbody>
</table>
// 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:
  
  $$a\text{.equals}(a) \text{ is true for every object } a$$
  $$a\text{.equals}(b) \iff b\text{.equals}(a)$$
  $$(a\text{.equals}(b) \land \land - \land \land c) \iff a\text{.equals}(c)$$

- No non-null object is equal to null:
  
  $$a\text{.equals}(null) \text{ is false for every object } a$$

- Two sets are equal if they contain the same elements:
  
  ```java
  Set<String> set1 = new HashSet<>();
  Set<String> set2 = new TreeSet<>();
  for (String s : "hi how are you".split(" ")) {
      set1.add(s);    set2.add(s);
  }
  System.out.println(set1.equals(set2));    // true
  ```
The hashCode method

```java
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:

```java
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 $T$.

  ```java
  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.

```java
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.

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