Object Oriented Programming

1. Class `String`
3. Operators, again
4. Control structures in Java
5. Classes and Objects

The Class `String`

- There is no primitive type for strings in Java
- The class `String` is a predefined class in Java that is used to store and process strings
- Objects of type `String` are made up of strings of characters that are written within double quotes
  - Any quoted string is a constant of type `String`

```
"Be happy learning Java."
```

- A variable of type `String` can be given the value of a `String` object

```
String blessing = "Be happy learning Java.";
```

Concatenation of Strings

- *Concatenation*: Using the + operator on two strings in order to connect them to form one longer string
  - If `greeting` is equal to `"Hello "`, and `javaClass` is equal to `"class"`, then `greeting + javaClass` is equal to `"Hello class"`
- Any number of strings can be concatenated together
- When a string is combined with almost any other type of item, the result is a string
  - `"The answer is " + 42` evaluates to

    `"The answer is 42"`

String Methods

- The `String` class contains many useful methods for string-processing applications
  - A `String` method is called by writing a `String` object, a dot, the name of the method, and a pair of parentheses to enclose any arguments
  - If a `String` method returns a value, then it can be placed anywhere that a value of its type can be used

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " +
  greeting.length());
```

  - Always count from zero when referring to the *position* or *index* of a character in a string

Some Methods in the Class `String`

Length

`i = ` *`s`*`.`**`length()`**	length of the string *s*.

Comparison (note: use these instead of == and !=)

`i = ` *`s`*`.`**`compareTo(`***`t`*`)`	compares to s. returns <0 if s<t, 0 if ==, >0 if s>t
`i = ` *`s`*`.`**`compareToIgnoreCase(`***`t`*`)`	same as above, but upper and lower case are same
`b = ` *`s`*`.`**`equals(`***`t`*`)`	true if the two strings have equal values
`b = ` *`s`*`.`**`equalsIgnoreCase(`***`t`*`)`	same as above ignoring case
`b = ` *`s`*`.`**`startsWith(`***`t`*`)`	true if *s* starts with *t*
`b = ` *`s`*`.`**`startsWith(`***`t`*`, `*`i`*`)`	true if *t* occurs starting at index i
`b = ` *`s`*`.`**`endsWith(`***`t`*`)`	true if *s* ends with *t*

Some Methods in the Class `String`

Searching - Note: **All "indexOf" methods return -1 if the string/char is not found**

`i = s.indexOf(t)`	index of the first occurrence of String *t* in *s*.
`i = s.indexOf(t, i)`	index of String *t* at or after position *i* in *s*.
`i = s.indexOf(c)`	index of the first occurrence of char *c* in *s*.
`i = s.indexOf(c, i)`	index of char *c* at or after position *i* in *s*.
`i = s.lastIndexOf(c)`	index of last occurrence of *c* in *s*.
`i = s.lastIndexOf(c, i)`	index of last occurrence of *c* on or before *i* in *s*.
`i = s.lastIndexOf(t)`	index of last occurrence of *t* in *s*.
`i = s.lastIndexOf(t, i)`	index of last occurrence of *t* on or before *i* in *s*.

Some Methods in the Class **String**

Getting parts

c = *s*.**charAt(***i*)	char at position *i* in *s*.
s1 = *s*.**substring(***i*)	substring from index *i* to the end of *s*.
s1 = *s*.**substring(***i*, *j*)	substring from index *i* to BEFORE index *j* of *s*.

Creating a new string from the original

s1 = *s*.**toLowerCase()**	new String with all chars lowercase
s1 = *s*.**toUpperCase()**	new String with all chars uppercase
s1 = *s*.**trim()**	new String with whitespace deleted from front and back
s1 = *s*.**replace(***c1*, *c2*)	new String with all *c2*s replaced by *c1*s.

Some Methods in the Class `String`

Static Methods for Converting to String

`s = String.valueOf(x)`	Converts x to String, where x is *any* type value (primitive or object).
`s = String.format(f, x...)`	[Java 5] Uses format *f* to convert variable number of parameters, *x* to a string.

- Note that the list is not exhaustive.

String Processing

- A **`String`** object in Java is considered to be immutable, i.e., the characters it contains cannot be changed
- There is another class in Java called **`StringBuffer`** that has methods for editing its string objects
- However, it is possible to change the value of a **`String`** variable by using an assignment statement

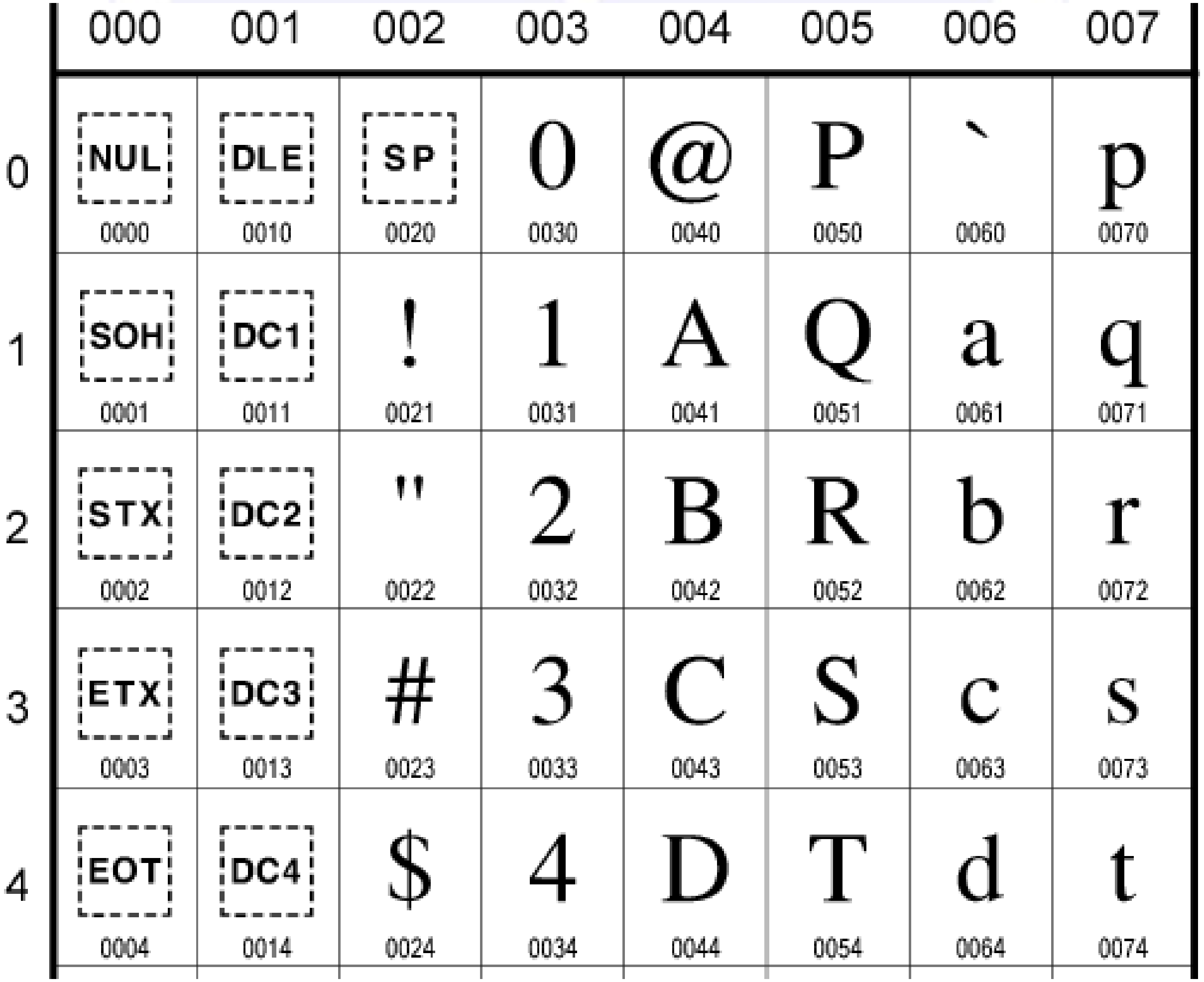
```
    String name = "Ionescu";
    name = "Ion " + name;
```

Character Sets

- *ASCII*: A character set used by many programming languages that contains all the characters normally used on an English-language keyboard, plus a few special characters
  - Each character is represented by a particular number

- *Unicode*: A character set used by the Java language that includes all the ASCII characters plus many of the characters used in languages with a different alphabet from English
  - Example: `char c='\u0103'; // Romanian letter 'ă'`

A small part of Unicode



	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074

Naming Constants

- Instead of using "anonymous" numbers in a program, always declare them as named constants, and use their name instead

```
public static final double CM_PER_INCH =
   2.54;
public static final int HOURS_PER_DAY = 24;
```

  - This prevents a value from being changed inadvertently
  - It has the added advantage that when a value must be modified, it need only be changed in one place
  - Note the naming convention for constants: Use all uppercase letters, and designate word boundaries with an underscore character

Comments

- A *line comment* begins with the symbols `//`, and causes the compiler to ignore the remainder of the line
  - This type of comment is used for the code writer or for a programmer who modifies the code
- A *block comment* begins with the symbol pair `/*`, and ends with the symbol pair `*/`
  - The compiler ignores anything in between
  - This type of comment can span several lines
  - This type of comment provides documentation for the users of the program

Program Documentation

- Java comes with a program called `javadoc` that will automatically extract documentation from block comments in the classes you define
  - As long as their opening has an extra asterisk (`/**`)
- Ultimately, a well written program is self-documenting
  - Its structure is made clear by the choice of identifier names and the indenting pattern
  - When one structure is nested inside another, the inside structure is indented one more level

Operators

- Are discussed in detail in the Laboratory Guide
- Some diferences from C:
  - bitwise operator >>>
    - E.g. n >>> p; // shifts the bits of *n* right *p* positions. Zeros are shifted into the high-order positions.
  - String concatenation operator +
  - Object operators – we'll discuss them in detail later

On Operator Precedence

<table>
<tr><th colspan="2">Operator Precedence</th></tr>
<tr><td>. [] (*args*) *post* ++ --
! ~ *unary* + - *pre* ++ --
(*type*) new
* / %
+ -
&lt;&lt; >> >>>
< <= > >= instanceof
== !=
&amp;
^
|
&amp;&amp;
||
?:
= += -= etc</td><td>Remember only
unary operators
* / %
+ -
comparisons
&amp;&amp; ||
= *assignments*
Use () for all others</td></tr>
</table>

The **if** Statement

- The **if** statement specifies which block of code to execute, depending on the result of evaluating a test condition called a *boolean expression*.

```
if (<boolean expression>)
  <then block>
else
  <else block>
```

- The `<boolean expression>` is a conditional expression that is evaluated to either true or false.
  - similar to C syntax, but remember what a boolean expression means in Java

Comparing Objects

- When two ***variables*** are compared, we are comparing their *contents*.
- In the case of ***objects***, the *content* is the *address* where the object is stored.
  - Note that strings in Java are objects of class String
  - Class String provides comparison methods as we already know
- The best approach for comparing objects is to *define comparison methods* for the class.

Hints for **if** code

- Start with the nominal case
  - Makes the code easier to read
- Don't forget the else clause!
- Avoid complicated conditions
- Break out into boolean variables/functions
- Try to use positive conditions
- Example – prefer second vs. first variant

```
if (!node.isFirst() && node.value() != null)
   stmts1
else      stmts2
if (node.isFirst() || node.value() == null)
   stmts2
else      stmts1
```

Hints for **if** chains

- All conditions should be closely related
- Put common cases first (when appropriate)
- Use **`switch`** if possible

```
// Good code
if (rnd < 0) {
// Error!
} else if (rnd < 0.1) {
// ...
} else if (rnd < 0.5) {
// ...
} else if (rnd < 1.0) {
// ...
} else
// Error!
```

```
// Bad code
if (screen.needsRepaint()) {
   // repaint screen
} else if (player1.canMove())
   {
   // get move from p1
} else if (player2.canMove())
   {
   // get move from p2
} else {
   // stalemate!
}
```

The **switch** Statement

- The syntax for the **switch** statement is

```
switch ( < expression>){
  <case label 1>: <case body 1>
  ...
  <case label n>: <case body n>
}
```

- The data type of `< expression>` must be **char, byte, short, int** or **String literal**
- The value of `<expression>` is compared against the constant *i* of `<case label i>`.

String literals in switch

```
public static void main(String[] args) {
  for (String argument : args) {
    switch (argument) {
      case "-verbose":
         case "-v":
           verbose = true;
           break;
         case "-log":
           logging = true;
           break;
         case "-help":
           displayHelp = true;
           break;
         default:
           System.out.println("Illegal command line argument");
        }
   }
   displayApplicationSettings();
}
```

The **switch** Statement

- If there is a matching case, its case body is executed. Otherwise, the execution continues to the statement following the **switch** statement
- The **break** statement causes execution to skip the remaining portion of the **switch** statement and resume execution following the **switch** statement.
- The **break** statement is necessary to execute statements in one and only one case.
  - Again, as in C

Hints for **switch**

- Order cases (logically or alphabetically)
  - Always have a `default` case
  - Always `break` cases
  - Try to keep the `switch` small
  - Break out large cases into functions

```
switch (file.getType()) {

  // Non-breaking case
  case IMAGE_PNG:
  case IMAGE_JPG:
  openWithPaint(file);
  break;

case IMAGE_WMF:
  displayWMF(file);
  break;
default:
  // Unknown type
  break;
}
```

Repetition Statements

- *Repetition statements* control a block of code to be executed for a fixed number of times or until a certain condition is met.
- Like C, Java has three repetition statements:
  - **while**
  - **do-while**
  - **for**
- Repetition statements are also called *loop statements*, and the `<statement>` part in what follows is known as the *loop body*.

The **while** Statement

- In Java, **while** statements follow a general format:

```
while ( <boolean expression> )
 <statement>
```

- As long as the `<boolean expression>` is true, the loop body is executed.
- In a *count-controlled loop*, the loop body is executed a fixed number of times.
- In a *sentinel-controlled loop*, the loop body is executed repeatedly until a designated value, called a *sentinel*, is encountered.

Pitfalls in Writing Repetition Statements

- With repetition statements, it is important to ensure that the loop will eventually terminate.
- Types of potential programming problems we should keep in mind:
- **Infinite loop**

```
int item = 0;
while (item < 5000) {
  item = item * 5;
}
```

  - Because **item** is initialized to 0, **item** will never be larger than 5000 (0 = 0 * 5), so the loop will never terminate.

Pitfalls in Writing Repetition Statements

- **Overflow error**

```
int count = 1;
while (count != 10)
{
  count = count + 2;
}
```

- In this example, (the **while** statement of which is also an infinite loop), count will equal 9 and 11, but not 10.
- An **overflow error** occurs when you attempt to assign a value larger than the maximum value the variable can hold.

Pitfalls in Writing Repetition Statements

- **Overflow errors**
- In Java, an overflow does not cause program termination.
  - With types **float** and **double**, a value that represents infinity is assigned to the variable.
  - With type **int**, the value “wraps around” and becomes a negative value. The representation behaves like all numbers would be stored on a circle and maximum positive and minimum negative would be neighbors.
- Real numbers should not be used in testing or increment, because only an approximation of real numbers can be stored in a computer.
- The **off-by-one error** is another frequently-encountered pitfall.

Hints on **while** loops

- Use for more complicated loops
  - Avoid more than one exit point
  - Breaks are allowed (to avoid code duplication)

```
// Bad code
stmts_A
while (boolExp) {
  stmts_B
  stmts_A
}
```

```
// Good code
while (true) {
  stmts_A
  if (!boolExp)
      break;
  stmts_B
}
```

The **do-while** Statement

- The **while** statement is a *pretest loop* (the test is done before the execution of the loop body). Therefore, the loop body may not be executed.
- The **do-while** statement is a *posttest loop*. With a posttest loop statement, the loop body is executed at least once.
- The format of the **do-while** statement is:

```
do
 <statement>
while (<boolean expression>);
```

- The `<statement>` is executed until the `<boolean expression>` becomes false.

Loop-and-a-Half Repetition Control

- Be aware of two concerns when using the loop-and-a-half control:
  - **The danger of an infinite loop.** The boolean expression of the `while` statement is true, which will always evaluate to true. If we forget to include an `if` statement to break out of the loop, it will result in an infinite loop.
  - **Multiple exit points.** It is possible, although complex, to write a correct control loop with multiple exit points (`break`s). It is good practice to enforce the *one-entry one-exit control* flow.

The **for** Statement

- The format of the for statement is as follows:

```
for (<initialization>; <boolean expression>; <increment>)
      <statement>
```

- Example:

```
int i, sum = 0;
for (i = 1; i <=100; i++){
  sum += i;
}
```

- The variable `i` in the example statement is called a *control variable*. It keeps track of the number of repetitions.
- The `<increment>` can be by any amount.
- Again, as in C

Hints on **for** Loops

- Ideal when the number of iterations is known
  - Only one statement per section
  - Declare loop variable in loop header (minimizes scope and avoids crosstalk)
  - Don't ever change the loop variable in the body of the loop

break with a Label

- **break** is used in **loops** and **switch**
  - it has different meanings for the two
- **break** can also be followed by a label, L
  - tries to transfer control to a statement labeled with L
  - A break with a label always terminates abnormally; if there are no statements labeled with L, a compilation error occurs
  - A labeled break lets you break out o multiple nested loops
  - The label must precede the outermost loop out of which you want to break
  - This form does not exist in C

break with a Label Example

```
int n;
read_data:
while(…) {
  …
  for (…) {
      n= Console.readInt(…);
      if (n < 0) // can't continue
            break read_data; // break out of data loop
      …
  }
}
// check for success or failure here
if (n < 0) {
  // deal with bad situations
}
else {
  // got here normally
}
```

continue with a Label

- The labeled form of the continue statement skips the current iteration of an outer loop marked with the given label.
- The label must precede the outermost loop out of which you want to break

continue with a Label – Sun's Example

```
public class ContinueWithLabelDemo {
    public static void main(String[] args) {
        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() - substring.length();
    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++)
                        != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
                    break test;
        }
        System.out.println(foundIt ? "Found it" :  "Didn't find it");
    }
}
```

for Statement for Iterating over Collections and Arrays

- Created especially for iterating over collections and arrays (we'll come back to it later) – Java 5
- Does not work everywhere (e.g. for access to array indices, it doesn't)
- Sun's example:

```
public class ForEachDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12,
                              1076, 2000, 8, 622, 127 };

        for (int element : arrayOfInts) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

General Hints for Loops

- Run the loop in your head (check end cases)
- Use meaningful names
  - Only use names such as `i, j, n` in short loops where the loop variable is just an index
- Avoid more than three nested loops (goes for ifs as well)
  - Restructure or break out helper functions
- Don't use the loop variable *after the end* of the loop

Statements for Processing Exceptions

- Just their meaning (in detail, later)
  - **`throw`** – throws an exception
  - **`try-catch`**, and **`finally`** – used to process exceptions
  - **`try`** – identifies a block of statements within which an exception might be thrown.
  - **`catch`** – associated with a **`try`** statement; identifies a block of statements that can handle a particular type of exception; the block is executed if an exception of a particular type occurs within the **`try`** block.
  - **`finally`** - associated with a try statement; identifies a block of statements that are executed regardless of whether or not an error occurs within the try block.
- Exceptions should never be used to simulate a goto!

Anatomy of a Class

```
public class Taxi{
  private int km;

  public Taxi(){
    km = 0;
  }

  public int getKm(){
    return km;
  }
   public void drive(int km){
    this.km += km;
  }
}
```

Class header

Instance variables (fields)

Constructors

Methods

A Constructor:

Purpose	Initialize an object’s state
Name	Same as its class. Upper case first letter, “camelcase” inside

Code	`public Taxi(){`
	…
	`}`

Output	No return type in header
Input	0 or more parameters

Usage	`> Taxi cab;`
	`> cab = new Taxi();`

# calls	At most once per object; Invoked by “new” operator

A Class with Multiple Constructors

```
public class Taxi{
 private int km;
 private String driver;

 public Taxi(){
    km = 0;
    driver = “Unknown”;
 }

 public Taxi(int km,String
d){
    this.km = km;
    driver = d;
 }
```

A successful “new” operation creates an object on the heap and executes the constructor whose parameter lists “matches” its argument list (by number, type, order).

```
> Taxi cab1;
> cab1 = new Taxi();
```

```
> Taxi cab2;
> cab2 = new
Taxi(10,”Jim”);
```

Proper use of constructors

- A constructor should *always* create its objects in a *valid* state
  - A constructor should not do anything *but* create objects
  - If a constructor cannot guarantee that the constructed object is valid, it should be private and accessed via a factory method

Proper use of constructors

- A factory method is a static method that calls a constructor
  - The constructor is usually private
  - The factory method can determine whether or not to call the constructor
  - The factory method can throw an Exception, or do something else suitable, if it is given illegal arguments or otherwise cannot create a valid object
  -
```
public static Person create(int age) {  // example factory
method
    if (age < 0) throw new IllegalArgumentException("Too
young!");
    else return new Person(age);
}
```

A Method:

Purpose	Execute object behavior
Name	A verb; Lower case with “camelcase”

Code	`public void turnLeft(){`
	`    …`
	`}`

Output	Return type required
Input	0 or more parameters
Usage	`> cab.turnLeft();`
# calls	Unlimited times per object

What Can a Method Do?

- A method can:
  - Change its object’s state
  - Report its object’s state
  - Operate on numbers, text, files, graphics, web pages, hardware, …
  - Create other objects
  - Call another object’s method: ***`obj.method(args)`***
  - Call a method in the same class: ***`this.method(args);`***
  - Call itself (recursion)
  - …

Method Declaration

```
public return_type methodName(0+ parameters){..}
public int getKM() {..}
public void increaseSpeed(int accel, int
limit){..}
```

- **Name:** Verb starting with a lowercase letter, with “camel caps”
- **Output:** Return type required.
- **Input:** 0 or more parameters
- **Body:**
  - Enclosed by curly braces
  - Contains an arbitrary # of statements (assignment, “if”, return, etc.).
  - May contain “local variable” declarations
- **How it’s called:** “dot” operator:

```
objectName.methodName(arguments)
cab1.increaseSpeed(5, 50)
```

Accessor and Mutator Methods

```
public class Taxi{
  private int km;

  public Taxi(){
    km = 0;
 }

  // gets (reports) # km
  public int getKm(){
    return km;
  }

 // sets (changes) # km
 public void setKm(int m){
    km = m;
 }
}
```

Accessor(aka getter)/ Mutator(aka setter) method calls

```
> Taxi cab;
> cab = new Taxi();
> cab.getKm()
0
> cab.setKm(500);
> cab.getKm()
500
```

A Method’s Input

- A method may receive 0 or more inputs.
- A method specifies its expected inputs via a list of “formal parameters” `(type1 name1, type2 name2, …)`
- In a method call, the number, order, and type of arguments must match the corresponding parameters.

Method Declaration (with parameters)	**Method Call (with arguments)**
`public void meth1(){..}`	`obj.meth1()`
`public int meth2(boolean b){..}`	`obj.meth2(true)`
`public int meth3(int x,int y,Taxi t){..}`	`obj.meth3(3,4,cab)`

Method Output

- A method may output nothing (void) or one thing.
- If it outputs nothing:
  - Its return type is “void”

    ```
    public void setKm(int km){..}
    ```
- If it outputs one thing:
  - Its return type is non-void (e.g. int, boolean, Taxi)

    ```
    public int getkm(){..}
    ```
  - It must have a return statement that returns a value

    ```
    // value returned must match return type
    return km;
    ```

Access Modifiers

- **public**- most often methods are given public access; everyone sees it
- **private** – can’t be used by all classes; seen only inside class
- **protected** – can’t be used by all classes; seen only inside the package
- **static** – objects aren’t required in order for these methods to be used
  - If the method declaration includes the **static** modifier, it is a class method.
  - Class methods can access only class variables and constants

Instantiable Class

- A class is *instantiable* if we can create instances of the class.
- Examples: wrapper classes for primitive integers, **String, Scanner,…** classes are all instantiable classes, while the **Math** class is not.
- Every object is a member of a class
  - Your desk is an object and is a member of the Desk class
  - These statements represent is-a relationships

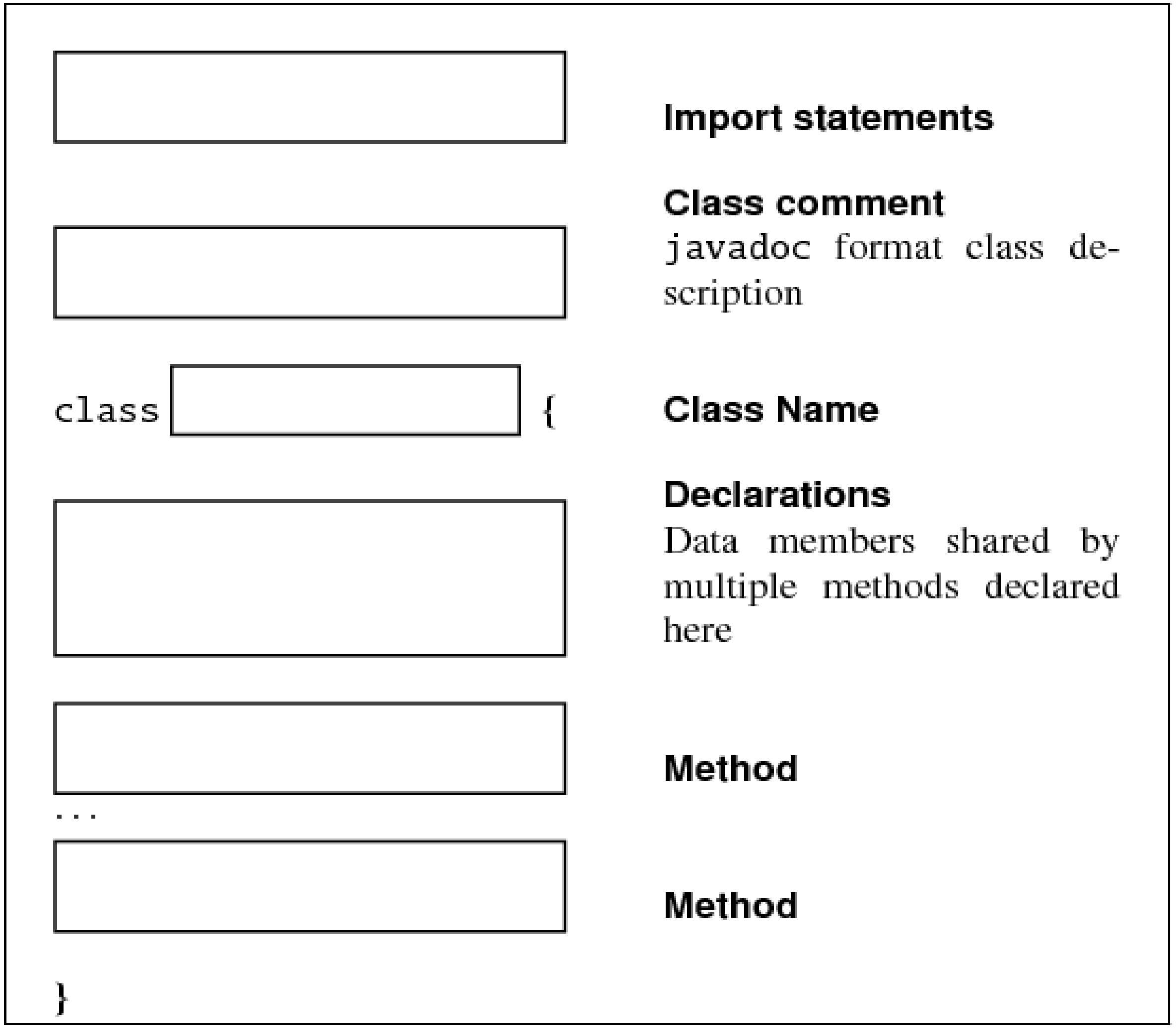
Utility of the Class Concept

- The concept of a class is useful because:
  - Objects inherit attributes from classes
  - All objects have predictable attributes because they are members of certain classes
- You must:
  - Create the classes of objects from which objects will be instantiated (e.g.Taxi class)
  - Write other classes to use the objects (a program/class is written to drive to the airport & uses the Taxi class to create a taxi object to drive)

Creating a Class

- You must:
    - Assign a name to the class
    - Determine what data and methods will be part of the class
- Class access modifiers include:
    - **public**
        - This is the most used modifier
        - Most liberal form of access
        - Can be **extended** or used as the basis for other classes
    - **final**- used only under special circumstances (class completely defined and no subclasses are to be derived from it)
    - **abstract**- used only under special circumstances (class is incompletely defined – contains an method which is not implemented)
        - We'll discuss the last two later

A Program Template for Class Code



class {

}

...

Import statements

Class comment
javadoc format class description

Class Name

Declarations
Data members shared by multiple methods declared here

Method

Method

Blocks and Scope

- **Blocks**: within any class or method, the code between a pair of curly braces
- The portion of a program within which you can reference a variable is its **scope**
- A variable comes into existence, or comes into scope, when you declare it
- A variable ceases to exist, or goes out of scope, at the end of the block in which it is declared
- If you declare a variable within a class, and use the same variable name to declare a variable within a method of the class, then the variable used inside the method takes precedence, or **overrides**, the first variable

Overloading a Method

Overloading:

- Involves using one term to indicate diverse meanings
- Writing multiple methods with the same name, but with *different arguments*
- Overloading a Java method means you write multiple methods with a shared name
- Example:

```
public int test(int i, int j){
    return i + j;
}
public int test(int i, byte j){
    return i + j;
}
```

Ambiguity with Overloading

- When you overload a method you run the risk of **ambiguity**
- An ambiguous situation is one in which the compiler cannot determine which method to use
- Example:

```
public int test(int units, int pricePerUnit)
{
    return units * pricePerAmount;
}
public long test(int numUnits, int weight)
{
    return (long)(units * weight);
}
```

Sending Arguments to Constructors

- Java automatically provides a constructor method when you create a class
- Programmers can write their own constructor classes, including constructors that receive arguments
  - Such arguments are often used for initialization purposes when values of objects might vary
- Example:

```
class Customer {
 private String name;
 private int accountNumber;
 public Customer(String n) {
        name =n;
 }
 public Customer(String n, int a) {
        name =n;
        accountNumber = a;
 }
}
```

Overloading Constructors

- If you create a class from which you instantiate objects, Java automatically provides a constructor
- But, if you create your own constructor, the automatically created constructor *no longer exists*
- As with other methods, you can *overload* constructors
  - Overloading constructors provides a way to create objects with or without initial arguments, as needed

The `this` Reference

- Classes can become large very quickly
  - Each class can have many data fields and methods
- If you instantiate many objects of a class, the computer memory requirements can become substantial
  - It is not necessary to store a separate copy of each variable and method for each instantiation of a class

The this Reference

- The compiler accesses the correct object’s data fields because you implicitly pass a **this** reference to class methods
- Static methods, or class methods, do not have a **this** reference because they have no object associated with them

```
 public getStudentID()
 {
   return studentID;
 }
 public getStudentID()
 {
   return this.studentID;
 }
```

Class Variables

- Class variables: variables that are shared by every instance of a class
- Company Name = "T.U. Cluj-Napoca"
- Every employee would work for the same company

```
private static String COMPANY_ID =
  "T.U. Cluj-Napoca";
```

  - It is possible but not recommendable to declare a variable that can be seen outside its class

Using Automatically Imported, Prewritten Constants and Methods

- The creators of Java created nearly 500 classes
  - For example, System, Character, Boolean, Byte, Short, Integer, Long, Float, and Double are classes
- These classes are stored in a package, or a library of classes, which is a folder that provides a convenient grouping for classes
- `java.lang` – The package that is implicitly imported into every Java program and contains fundamental classes, or basic classes
- Fundamental classes include:
  - System, Character, Boolean, Byte, Short, Integer, Long, Float, Double, String
- Optional classes – Must be explicitly named

Using Prewritten Imported Methods

- To use any of the prewritten classes (other than java.lang):
  - Use the entire path with the class name

```
area = Math.PI * radius * radius;
```

or

  - Import the class

or

  - Import the package which contains the class you are using
- To import an entire package of classes use the wildcard symbol *
- For example:
  - import java.util.*;
  - Represents all the classes in a package

Static Methods

- In Java it is possible to declare methods and variables to belong to a *class* rather than an *object*. This is done by declaring them to be **static**.
- Static methods are declared by inserting the word "static" immediately after the scope specifier (*public*, *private* or *protected*).

```
public class ArrayWorks {
   public static double mean(int[] arr)   {
            double total = 0.0;
            for (int k=0; k!=arr.length; k++)  {
                  total = total + arr[k];
            }
      return  total / arr.length;
   }
}
```

Static Methods

- Static methods are called using the name of their class in place of an object reference.

```
double myArray = {1.1, 2.2, 3.3};
...
double average = ArrayStuff.mean(myArray);
```

- Example of the utility of static methods: in the standard Java class, called **Math**.

```
public class Math {
  public static double abs(double d) {...}
  public static int abs(int k) {...}
  public static double pow(double b, double exp) {...}
  public static double random() {...}
  public static int round(float f) {...}
  public static long round(double d) {...}
  …
}
```

Static Method Restrictions

- The body of a static method cannot reference any non-static instance variable.
- The body of a static method cannot call any non-static method.
- But, the body of a static method can instantiate objects.

```
public class go {
   public static void main(String[] args)   {
      Greeting greeting = new Greeting();
   }
}
```

- Java standalone applications are required to initiate execution from a static void method that is always named *main* and has a single String array as its parameter.

Static Variables

- Any instance variable can be declared **static** by including the word “static” immediately before the type specification

```
public class StaticStuff {
    public static double staticDouble;
    public static String staticString;
    . . .
}
```

- A static variable:
  - Can be referenced either by its class or an object
  - Instantiating a second object of the same type does not increase the number of static variables.

Static Variables Example

```
StaticStuff s1, s2;
s1 = new StaticStuff();
s2 = new StaticStuff();
s1.staticDouble = 3.7;
System.out.println( s1.staticDouble );
System.out.println( s2.staticDouble );
s1.staticString = “abc”;
s2.staticString = “xyz”;
System.out.println( s1.staticString );
System.out.println( s2.staticString );
```

Why Static Methods and Variables?

- Static methods are useful for situations where data needs to be shared across multiple objects of the same type.
- A good example of the utility of static method is found in the standard Java class, called **Color**

```
public class Color {
   public static final Color black = new Color(0,0,0);
   public static final Color blue = new Color(0,0,255);
   public static final Color darkGray = new
                                 Color(64,64,64);
    . . .
}
```

- Color constants are both static and final => we can compare them

```
 Color myColor;
 ...
 if (myColor == Color.green) ...
```

Reading

- Deitel – chapters 3, 4, 5
- Eckel – chapters 4, 5
- Barner – chapters 1, 2

(see slide 8 of lecture 1 for full names of books)

Summary

- Class `String`
- Control statements
  - if, switch, while, for, do – while
  - break/continue with a label
  - for each
  - acces modifiers
  - Overloading
- Methods:
  - kinds (accessors, mutators)
  - input, output
- Class
  - constructors
  - instantiable class
  - creation, constructor overloading
  - the `this` reference
  - class variables
- Static
  - methods
  - variables