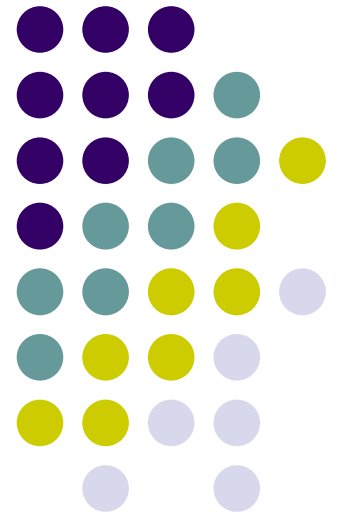


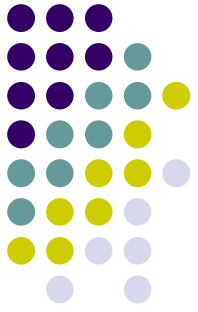
# Advanced Java Programming

---

## Objects & Classes

Ohad Barzilay,  
Tel-Aviv University  
Spring '06

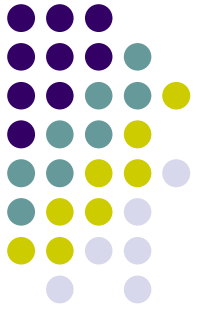




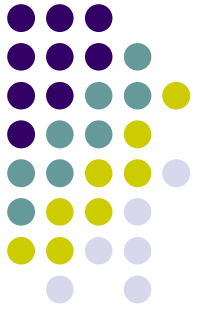
# Introduction to OOP

- One of the ideas of object-oriented software is to organize software in a way that matches the thinking style of our object-oriented brains.
- Of course, at the machine level nothing has changed - bit patterns are being changed by machine instructions. But we don't have to think about it that way.

# Characteristics of Objects



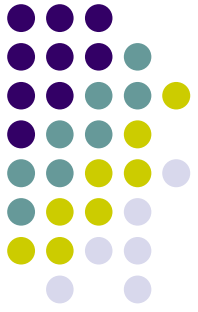
- An object has **identity** (it acts as a single whole).
- An object has **state** (it has various properties, which might change).
- An object has **behavior** (it can do things and can have things done to it).



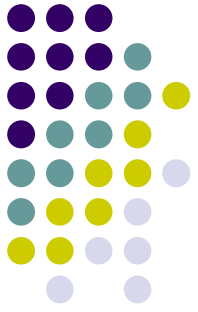
# Software Objects

- Many programs are written to do things that are concerned the real world. It is convenient to have "software objects" that are similar to "real world objects."
- Software objects will have *identity*, *state*, and *behavior* just as do real world objects.
- In this context we use the terms "problem domain" and "software domain"

# Objects

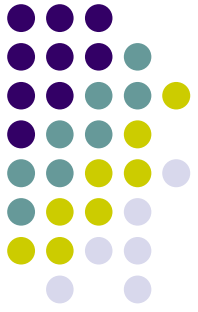


- Software objects have identity because each is a separate chunk of memory.
- Software objects have state. Some of the memory that makes a software object is used for variables which contain values.
- Software objects have behavior. Some of the memory that makes a software object is used to contain *methods* that enable the object to "do things."



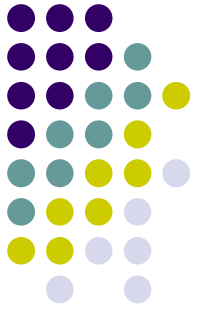
# Classes

- When a Java application is being run, objects are created and their methods are invoked.
- To create an object, there needs to be a description of it.
- A class is a description of a kind of object. A programmer may define a class using Java, or may use predefined classes that come in class libraries.



# Creating Objects

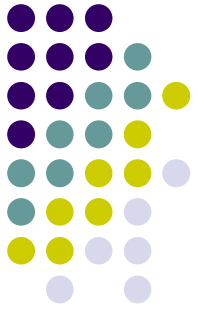
- A class is merely a plan for a possible object.
- When a programmer wants to create an object the `new` operator is used with the name of the class. Creating an object is called *instantiation*.



# Creating Objects – The new Operator

- `String name = new String("Yoav");`
- `Turtle leonardo = new Turtle();`
- `Point x = new Point (2,3);`

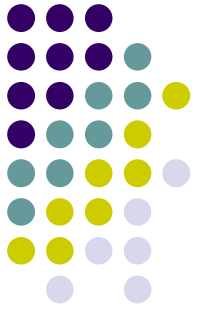




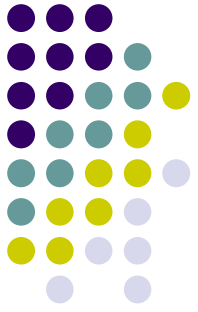
# Is Everything an Object?

- Java is almost pure object oriented. The only things which are not objects are primitive types (hence, the name):
  - `byte, short, int, long, float, double, char, boolean`
- However, Java includes special classes of similar names which are objects:
  - `Byte, Short, Integer, Long, Float, Double, Character, Boolean`

# Primitive Data Types and Classes



- Java has *many* data types built into it, and you (as a programmer) can create as many more as you want.
- However, other than the primitive data types, *all the other data in a Java program will be represented as an object.*



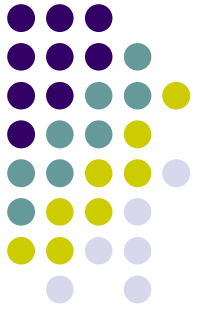
# Two kinds of variables

- **Primitive variable**

- Contains the actual data.
- Used for primitive data types

- **Reference variable**

- Contains information on how to find the object
- Used for all Objects



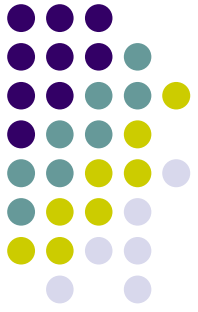
# Reference Variables and Null

- Since objects are big, complicated, and vary in size you do not automatically get an object when you declare an object reference variable.

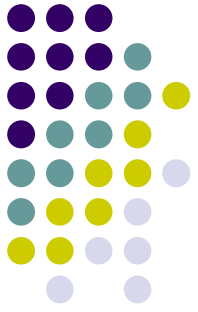
```
String name;
```

```
name = new String( "Dana" );
```

# Variables

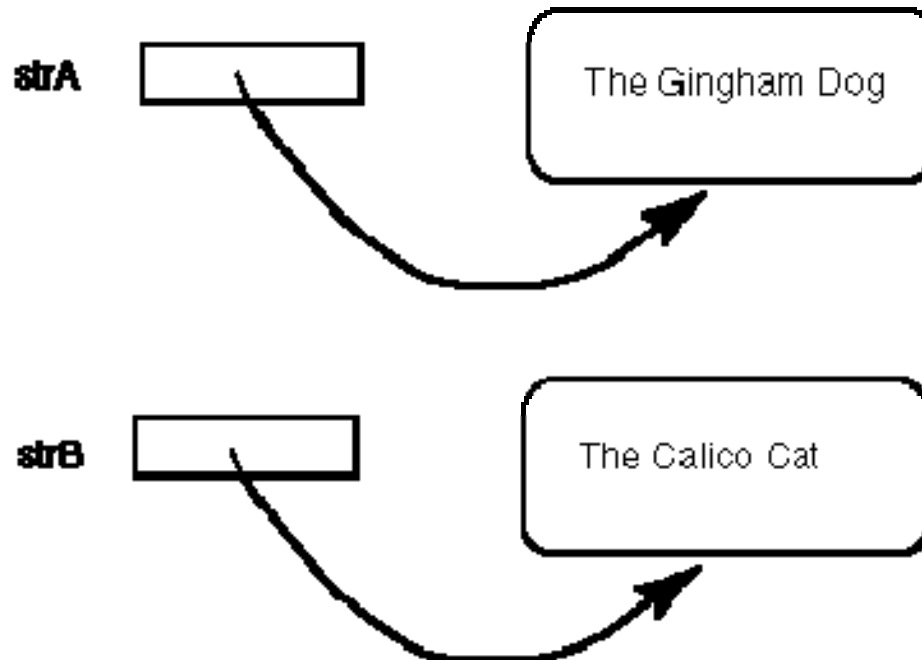


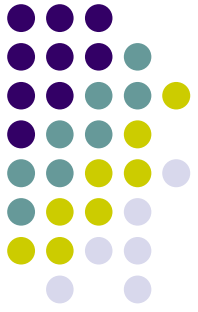
<b>Kind of Variable</b>	<b>Information it Contains</b>	<b>When on the left of "="</b>
<b>primitive variable</b>	Contains actual data.	Previous data is replaced with new data.
<b>reference variable</b>	Contains information on how to find an object.	Old reference is replaced with a new reference



# Several Objects of the same class

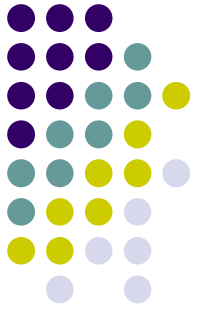
```
strA = new String( "The Gingham Dog" );  
strB = new String( "The Calico Cat" );
```





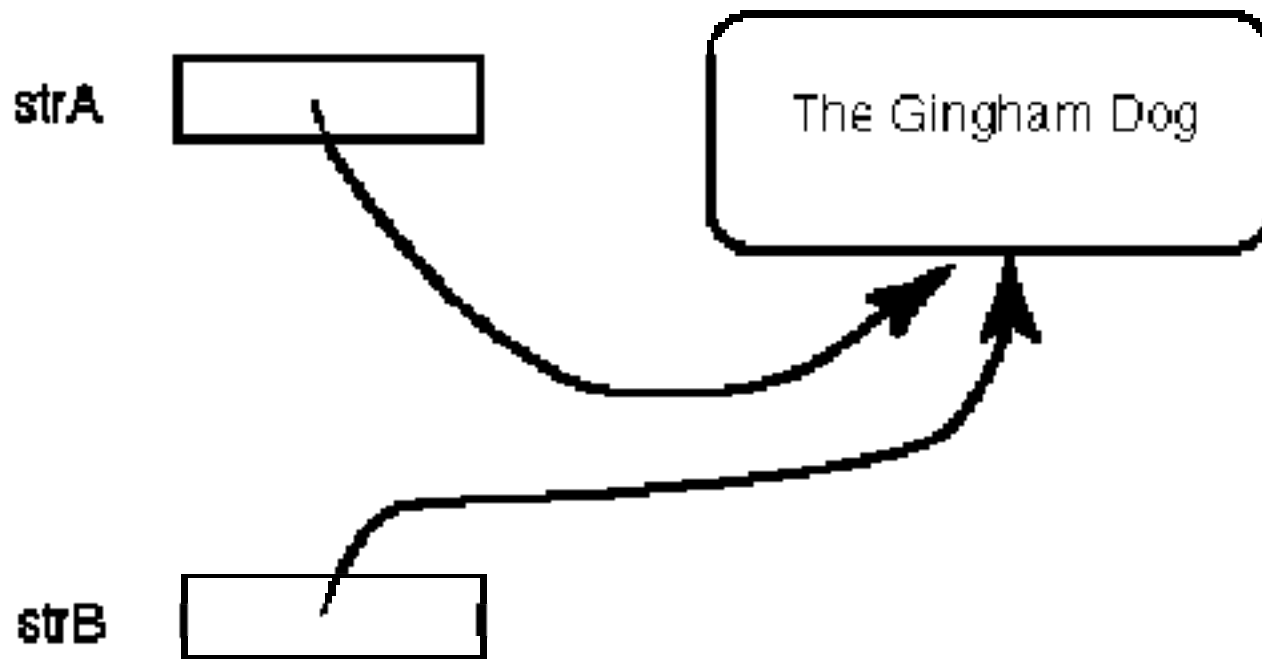
# Equality of reference

- The `==` operator is used to look at *the contents* of two reference variables. If the contents of both reference variables is the same, i.e, both variables refer to the same object then the result is *true*.
- The `==` operator *does **NOT** look at objects!* It only looks at references (information about where an object is)

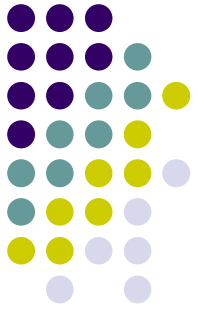


# Alias

```
strA = new String( "The Gingham Dog" );  
strB = strA;
```



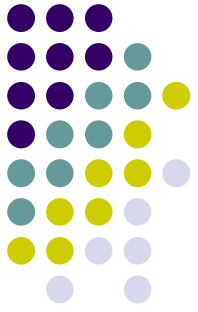




# The equals method

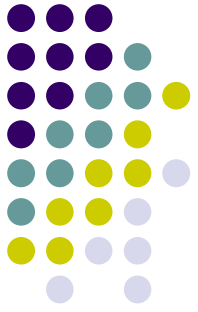
- The `equals()` method is used to determine if two objects contain the same data.
- Two Strings that are `==` are always `equals()`
- This holds for any 2 objects





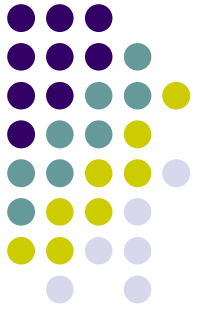
# Garbage Collection

- An Object which has no reference to it is called garbage.
- Since a garbage object can not be referred to, it is a waste of memory.
- The java virtual machine must clear that unused memory so it can be reused for creating other objects. This process is called ***Garbage Collection***



# Garbage Collection

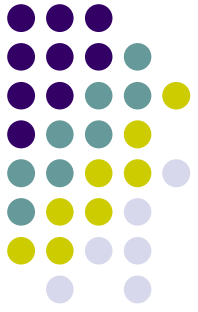
- Garbage collection takes care of freeing the memory storage.
- The basic idea is storage reuse:
  - When an object is created the memory space is allocated for this object.
  - Later if the object or data are not used the memory is returned to the system for future reuse.
- The JVM performs the garbage collection periodically automatically.



# Invoking an Objects method

- Remember that an object consists of methods (recipes for behavior.)
- Java uses "dot notation" for invoking methods. To invoke the `length()` method of the object named `name` the following is used:  

```
int len = name.length();
```

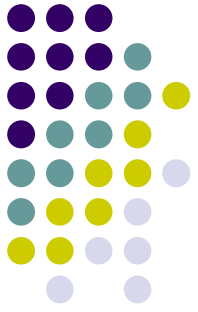


# Invoking an Objects method

```
Turtle leonardo = new Turtle();  
leonardo.moveForward(50);  
leonardo.turnRight(90);
```

```
Point p = new Point(0,0);  
p.translate(2,3);
```

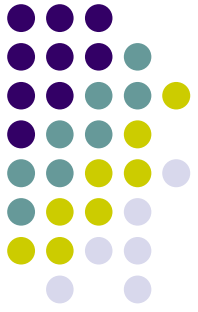
- How can we know which methods can be invoked for every object ?



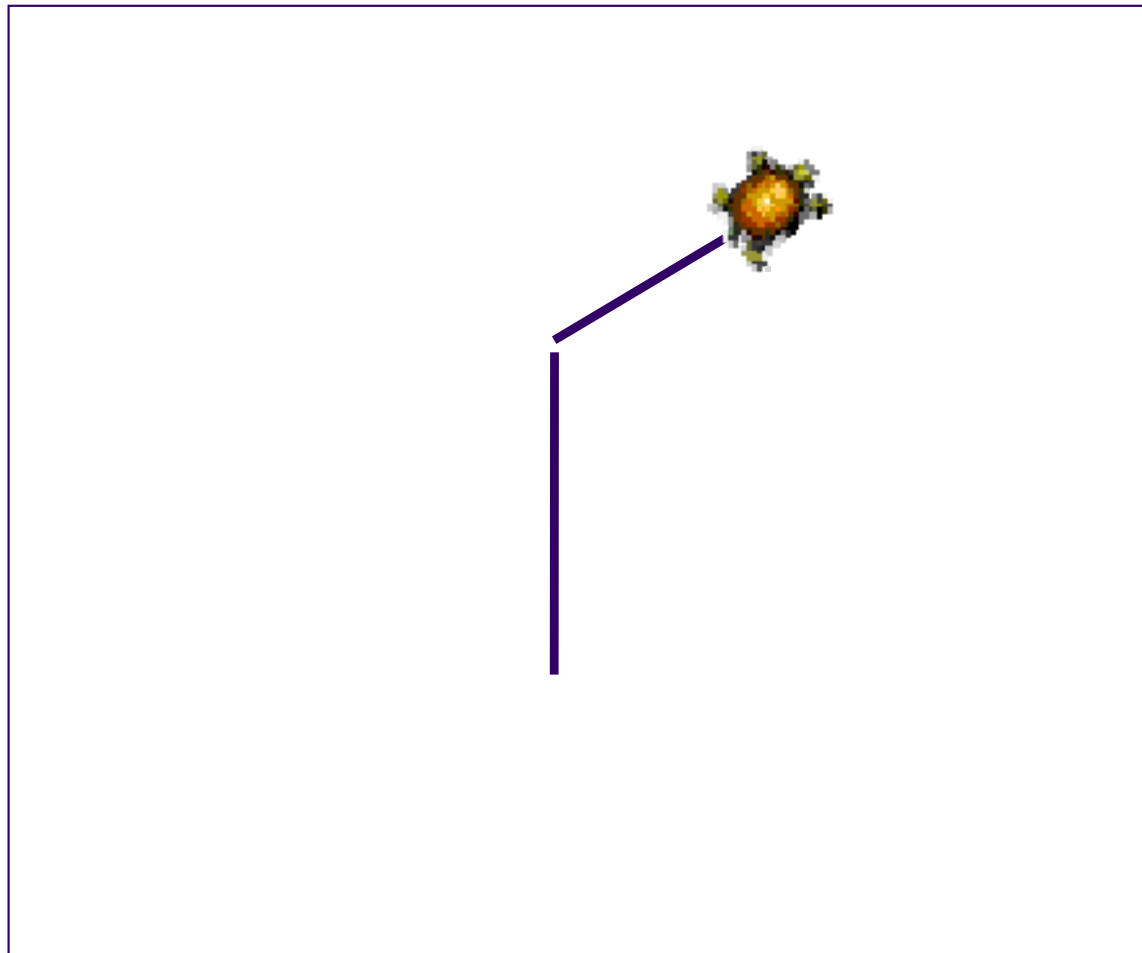
# API

(Application Programming Interface)

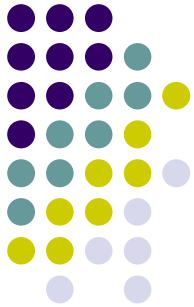
- The API is a Specification of methods and constructors in the class. It is the Documentation of the class.
- When we talk about defining are own classes we will see how to generate API documentation.



# LOGO Turtle



# Turtle Documentation



Turtle - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites Refresh Print Mail Print Preview

Address <D:\ohad\soft1\workspace\advJava\doc\Turtle.html> Go Links >>

Google Search 1073 blocked AutoFill Options

---

**Package** [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#) [All Classes](#)

[SUMMARY: NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) [DETAIL: FIELD](#) | [CONSTR](#) | [METHOD](#)

---

## Class Turtle

java.lang.Object  
└─ **Turtle**

---

```
public class Turtle
extends java.lang.Object
```

---

### Constructor Summary

[Turtle](#) ()

---

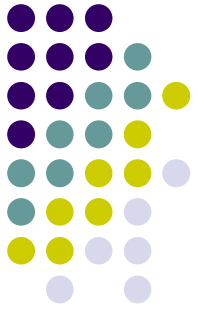
### Method Summary

void	<a href="#">hide</a> ()	Hides the turtle.
void	<a href="#">moveBackward</a> (double distance)	Moves the turtle backwards by a given number of units.
void	<a href="#">moveForward</a> (double distance)	Advances the turtle forwards by a given number of units.
void	<a href="#">show</a> ()	Shows the turtle.
void	<a href="#">tailDown</a> ()	Lowers the tail of the turtle.

Done My Computer

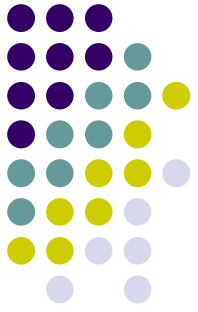
Start 4 Inte... 3 Wind... 02\_Defi... Java - T... Address <http://www.gmail.com/> Go EN << 11:39





# Turtle Methods

- `show()` *Shows the turtle.*
- `hide()` *Hides the turtle.*
- `moveBackward(double)` *Moves the turtle backwards by a given number of units.*
- `moveForward(double)` *Advances the turtle forwards by a given number of units.*
- `tailDown()` *Lowers the tail of the turtle.*
- `tailUp()` *Raises the tail of the turtle.*
- `turnLeft(int)` *Turns the turtle counter-clockwise.*
- `turnRight(int)` *Turns the turtle clockwise.*

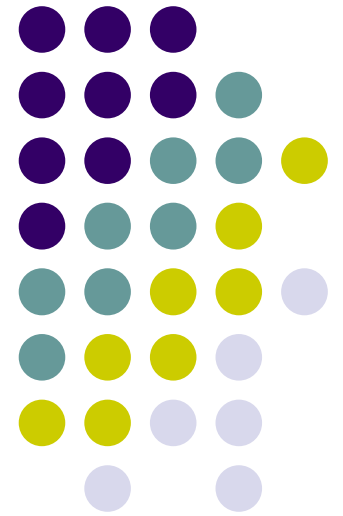


# Using the Turtle

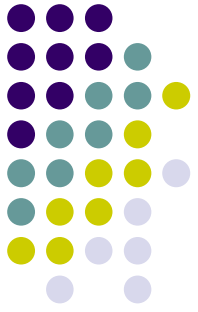
```
public class TurtleDrawing {  
    public static void main(String[] args) {  
        Turtle leonardo = new Turtle();  
        leonardo.tailDown();  
        leonardo.moveForward(100);  
        leonardo.turnLeft(90);  
        leonardo.moveForward(100);  
        // ...  
    }  
}
```

# Defining Classes

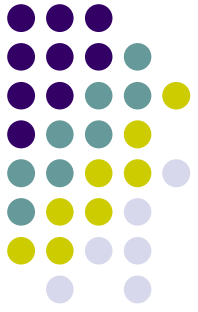
---



# Container vs. Definition Classes

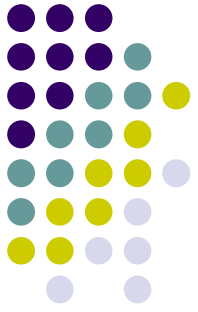


- Container classes:
  - A collection of static methods that are not bound to any particular object.
  - These static methods usually have something in common.
  - Class as *Module*
- Definition classes:
  - These classes define new objects, in a way that we will soon see.
  - Class as *Type*



# Classes As New Types

- The class declaration is a way of defining new types for your program – extending your language “vocabulary”.
- Once a class is declared you can use it to declare object variables.
- All those objects will be of the same type, they will have the same data and the same methods.



# Container Class

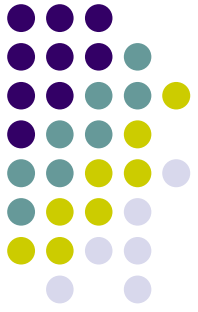
- The Math class is an example of the first kind. It is a container for math utility methods:

`Math.sqrt ( )`

`Math.abs ( )`

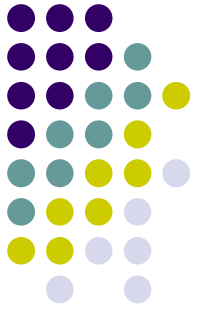
`Math.max ( )`

...



# Definition Class

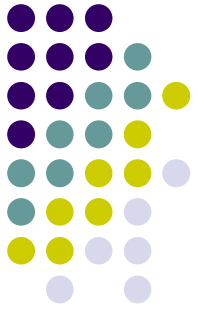
- The class *Turtle* is an example of the second kind. It defines a new type of objects, *Turtle* objects.
- We will focus more on the second kind.



# Defining Classes - Abstraction

- Think of the object in its ideal form: what does it represent? What behaviors it should have?
- It is often useful to think of a real-world comparison, to help choose the name for the class and its methods.
- Often we include methods that we don't immediately need only because they make sense for the abstraction of the object.

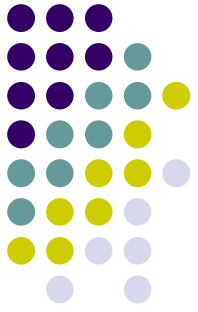




# Objects and classes

A class is build of 4 parts:

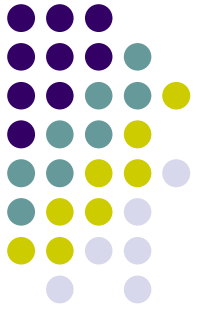
1. **Definition** - class name;
2. **Properties** - instance and class variables;
3. **Constructors** - special methods for creating objects
4. **Methods** - instance and class methods



# Objects and Classes

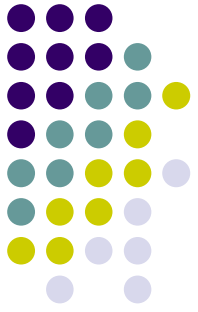
- A **class** is a description of a possible object. The description is provided in code, therefore we can only write classes.
- An **object** is a unique runtime instance of a class. Objects are located in the memory and are created only when the program is executed.

# Object Oriented Programming

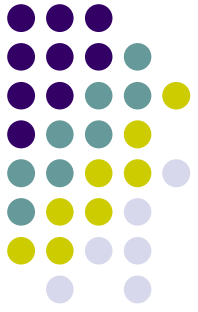


- The programmer defines classes that describe future objects that the program will use when it is running.
- As the program runs, The program does its work by creating objects and activating their methods.
- Each class description can be used to create many objects of the same type. A class is a blue print for creating objects.

# Syntax of class definition

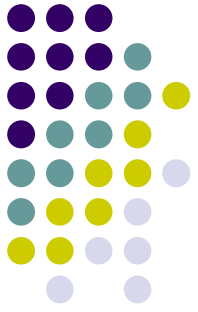


```
class ClassName {  
    // description of variables  
    // constructors  
    // methods  
}
```



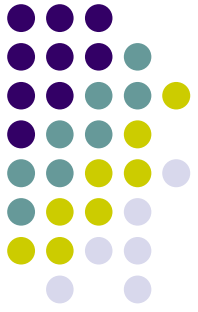
# Methods

- A Java class contains one or more methods.
- Each method is defined by its:
  - Visibility - (public / private / protected)
  - Return value (int / double / Point / void)
  - Arguments



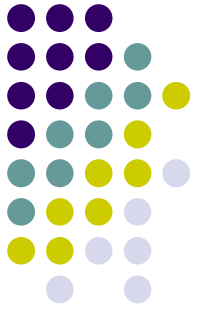
# Constructors

- Objects must be initialized before they can be used .We must specify what is the initial state of the object before we can use it.
- We specify the way an object is initialized using a constructor, which is a special method which is invoked every time we create a new object.



# Constructors

- Each class has a constructor. It is the first thing that is executed when creating an object.
- Constructors do not have a return type.
- The constructor has the same name as the class name.
- It is possible to have more than one constructor in a class, each constructor differs in its arguments.



# Calling the Constructor

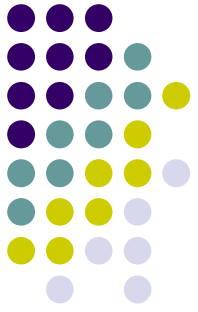
```
Point p1 = new Point();
```

```
Point p2 = new Point(1,1);
```

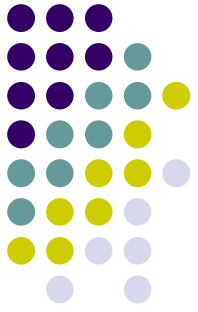
```
Point p3 = new Point(p2);
```



# Examples

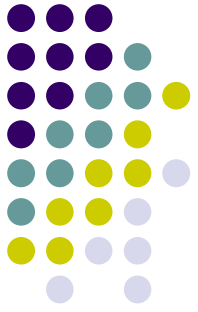


- Point
- Segment
- Complex
- LinearFunction
- Circle



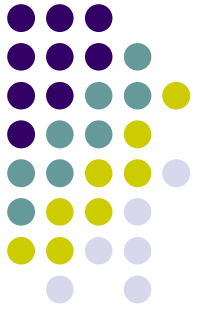
# Class Vs. Objects

- Most of the data and methods reside and work on specific objects of a class. The class is only a blueprint used as object definition.
- This view is not totally correct. There are situations in which data and/or methods reside in the class itself.



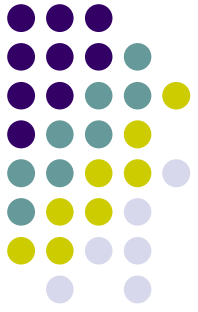
# Static Data Members

- Only one instance of a static variable exists for the whole class. The value of the variable is one for all instances.
- The variable exist even before any objects of the class are instantiated.
- By default static variables are initialized to 0 or null.



# Static Methods

- Static methods can operate only on static variables or call other static methods.
- They are not connected to a specific instance and therefore cannot use `this` reference.
- Instance methods can access static variables!



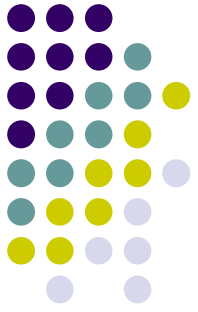
# Static Method Call

- The access to a static method is not done through an object but through the class name itself:

```
float i = Math.random();
```

```
String s = String.valueOf(i);
```

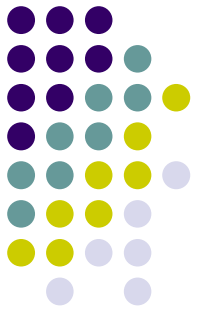
- If a static method calls another static method of the same class than the class-name can be omitted.



# The Main Method

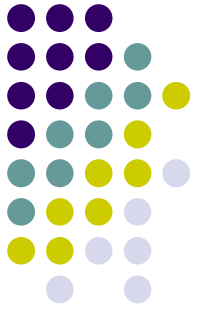
- The *main* method is static; it is invoked by the system without creating an instance object!

```
public static void main(String[] args);
```



# A Collection of Static Methods

- Even if no static variables exist static methods can still be used to implement general functionality. They just encapsulate a given task, a given algorithm.
- We can write a class that is a collection of static methods. Such a class isn't meant to define new type of objects.
- This class is more of a *namespace* or a *module*

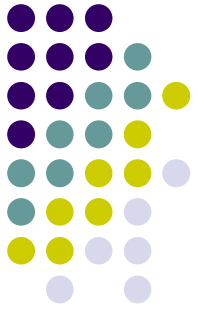


# java.lang.Math

```
/**
 * A library of mathematical methods.
 */
public class Math {
    // Computes the trigonometric sine of an angle.
    public static double sin(double x) {
        // ...
    }

    // Computes the logarithm of a given number.
    public static double log(double x) {
        // ...
    }
    // ...
}
```





# Use of Math Methods

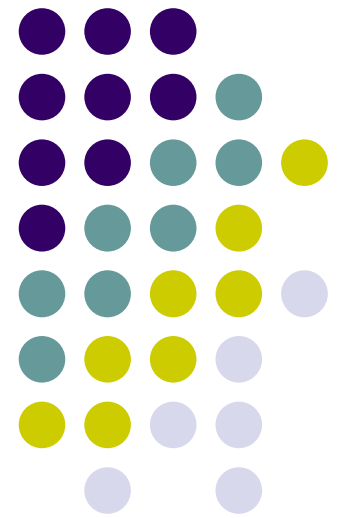
- It is just used as a library for utilities that are related in some way:

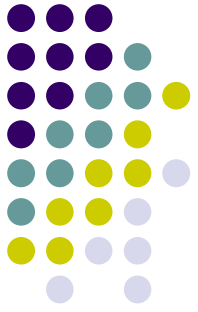
```
double x = Math.sin(alpha);  
int c = Math.max(a,b);  
double y = Math.random();
```

# API Documentation

---

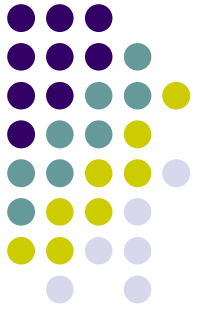
## Javadoc





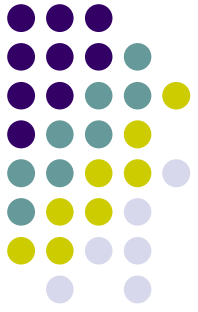
# Interface Definition

- Your classes are often intended to be used by other programmers. Even when we hide implementation, we need to describe what a class can do.
- Programmers that use your class are not interested in the implementation. They want to use it as a whole and are only interested in **what it does** and **how to use it**.



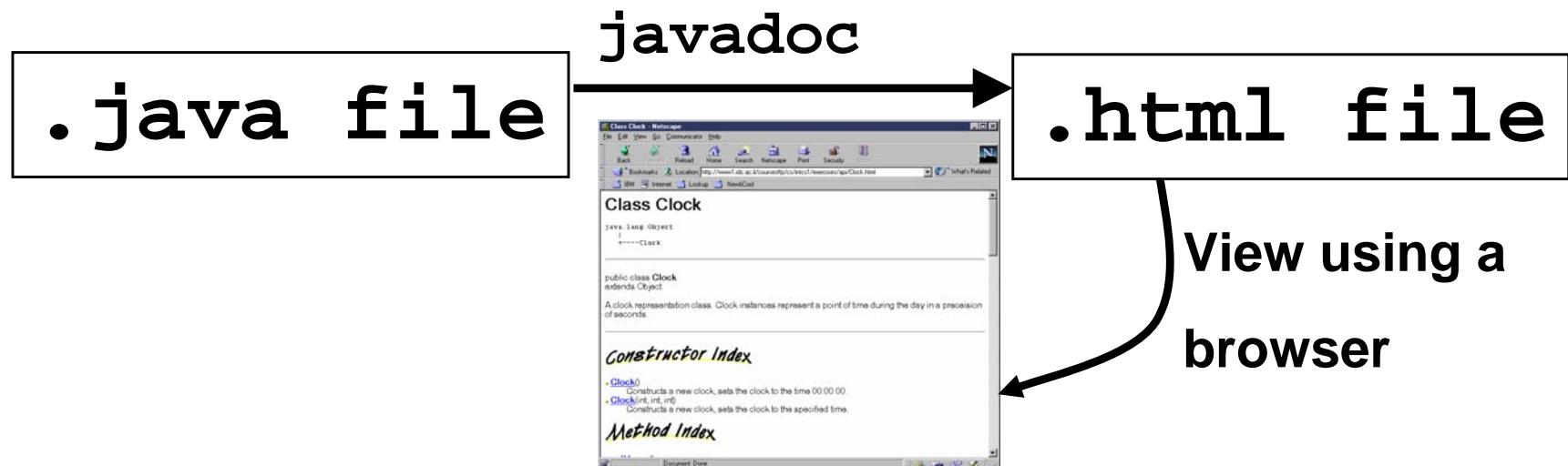
# API Documentation

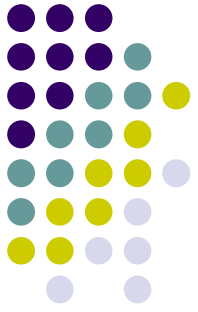
- API (Application Programmer Interface) documentation is a description of the interface of the class intended for the application programmer who wants to use it.
- To use the class, we need not (and should not) look at the code. All that is needed is the class API.



# Javadoc

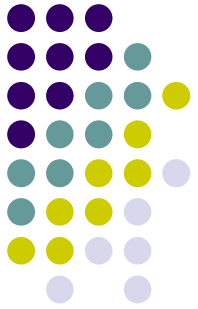
- The JDK contains a special tool for the generation of API documentation for your classes, called `javadoc`.





# Javadoc Process

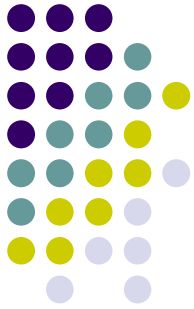
- `javadoc` takes as input Java programs and automatically generates documentation using:
  - The `public` method signatures
  - Any documentation which is part of the interface begins with `/**` (double asterisk) and ends with `*/`
- The output is an HTML file which can be viewed by an internet browser.



# From Source File

```
•/**
•* A clock representation class. Clock instances represent a point of
•* time during the day in a precision of seconds.
•*/
•public class Clock {
•    // the hours, minutes and seconds read
•    private int hours, minutes, seconds;
•    /**
•    * Constructs a new clock, sets the clock to the time 00:00:00.
•    */
•    public Clock() {
•        hours = 0;
•        minutes = 0;
•        seconds = 0;
•    }
•    /**
•    * Constructs a new clock, sets the clock to the specified time.
•    * @param hours The hours to be set (0-23)
•    * @param minutes The minutes to be set (0-59)
•    * @param seconds The seconds to be set (0-59)
•    */
•    public Clock(int hours, int minutes, int seconds) {
•        setTime(hours, minutes, seconds);
•    }
•    // ...
•}
```

javadoc



# To HTML Output

## Class Clock

```
java.lang.Object
|
+--Clock
```

```
public class Clock
    extends java.lang.Object
```

A clock representation class. Clock instances represent a point of time during the day in a precision of seconds.

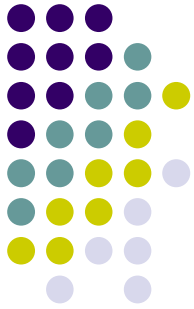
### Field Summary

protected int	<a href="#">hours</a>
protected int	<a href="#">minutes</a>
protected int	<a href="#">seconds</a>

### Constructor Summary

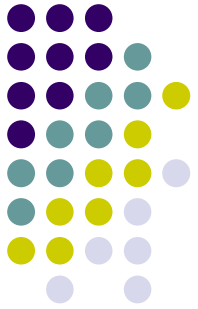
<a href="#">Clock()</a>	Constructs a new clock, sets the clock to the time 00:00:00.
<a href="#">Clock(int hours, int minutes, int seconds)</a>	Constructs a new clock, sets the clock to the specified time.



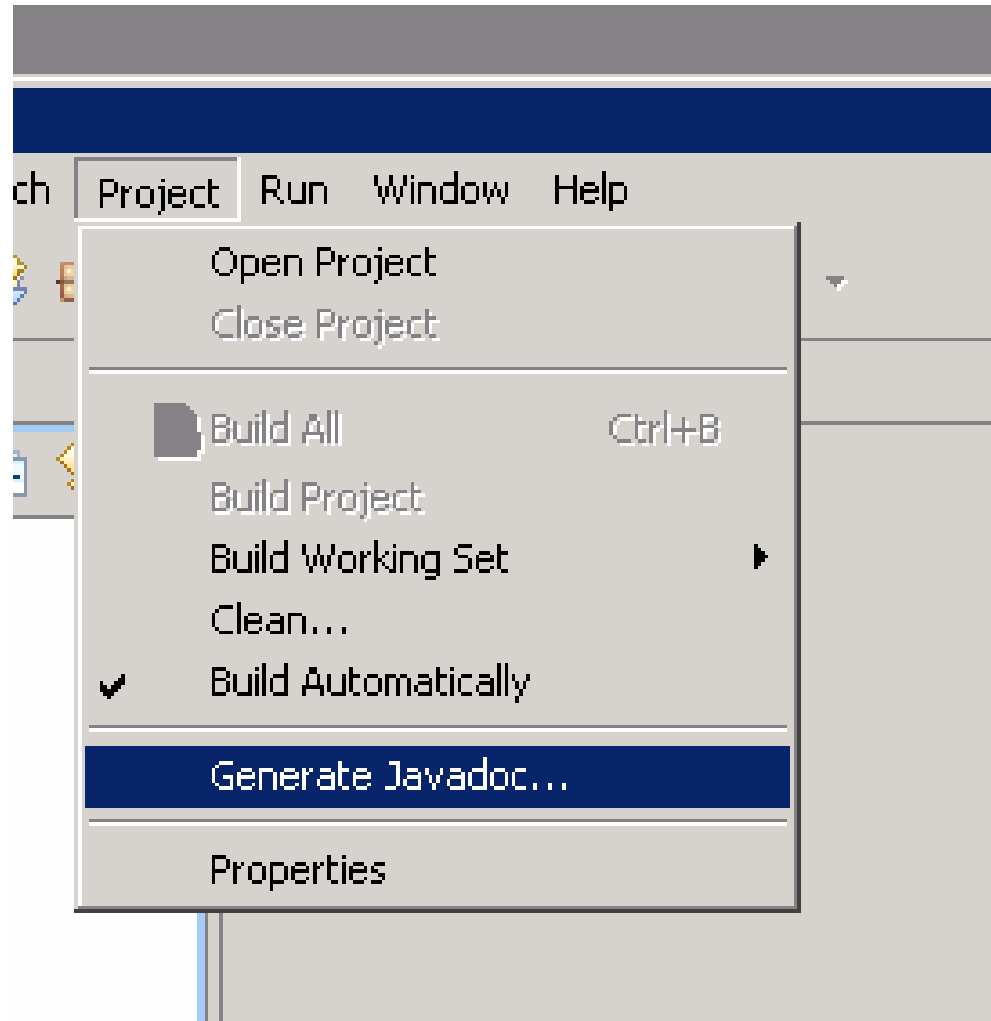


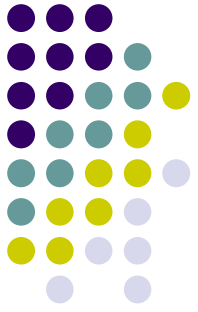
# Clock Class Documentation

Method Summary	
<code>boolean</code>	<a href="#"><code>equals</code></a> (Clock compareClock) Returns true if parameter clock is equal to this clock.
<code>int</code>	<a href="#"><code>getHours</code></a> () Returns the hours read
<code>int</code>	<a href="#"><code>getMinutes</code></a> () Returns the minutes read
<code>int</code>	<a href="#"><code>getSeconds</code></a> () Returns the seconds read
<code>void</code>	<a href="#"><code>hourElapsed</code></a> () Advances the clock by one hour.
<code>void</code>	<a href="#"><code>minuteElapsed</code></a> () Advances the clock by one minute.
<code>void</code>	<a href="#"><code>secondElapsed</code></a> () Advances the clock by one second.
<code>void</code>	<a href="#"><code>setTime</code></a> (int hours, int minutes, int seconds) Sets the clock to the specified time.
<code>java.lang.String</code>	<a href="#"><code>toString</code></a> () Returns a string description of the time represented by the clock in the format hh:mm:ss.



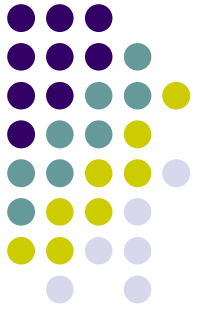
# Directly from Eclipse





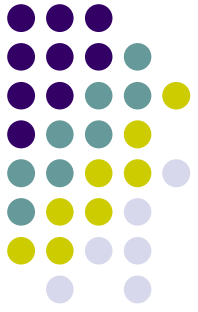
# What Should You Comment?

- You should put a documentation comment for any member of the class which is part of its interface and for the class itself.
- All public constructors and methods should documentation comments.
- Private methods are not part of the interface of the class, thus `javadoc` skips them by default
- You may ask `javadoc` to include also non-public parts of the class



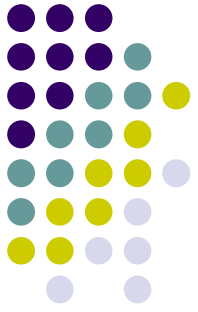
# API Documentation Purpose

- Documentation comments are written for programmers who use your class as a whole.
- They should describe only
  - What the class does,
  - How to use it.



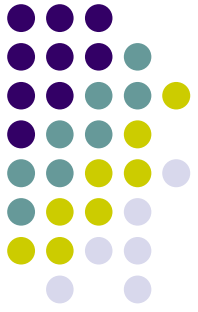
# API Documentation Style

- Documentation comments should not describe how a class is implemented.
- Documentation comments should be
  - Short and descriptive,
  - Written in a simple language (ENGLISH),
  - Accurate.
- Assume that the reader doesn't know anything about your class



# API Documentation Tags

- Documentation comments can also include *tagged paragraphs* that give a standard way to document several features of the interface such as method parameters, return values, etc.
- A tagged paragraph begins with the symbol @ followed with a tag keywords. Tags: **@see**, **@author**, **@version**, **@param**, **@return**, **@exception**.
- Documentation comments text can include HTML tags (not relevant for those who don't know HTML).

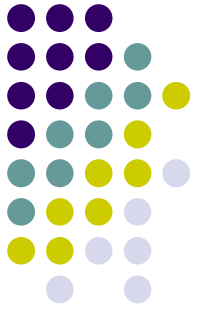


# Using Tags

- `@param` to document every parameter in the method signature: description, range limitations, etc.
- `@return` if the method returns a value:

```
/**
 * Reads the next line in the file.
 * @return A string containing the next line in the file,
 * not including newline character at end (if exists).
 */
public String nextLine() {
    // ...
}
```

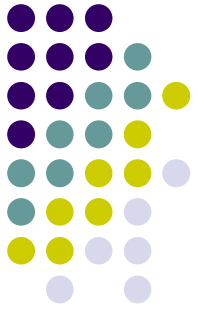
- `@exception` to document exceptions (later).



# Naming

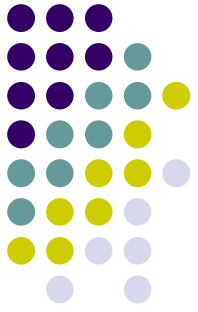
- The names you use for your class and for its public methods are part of the class API.
- Good descriptive naming are crucial for a clear API.
- Read the style guidelines!





# Changing the Implementation

- Encapsulation allows us to change an implementation of a class without affecting other parts of the program.
- Without encapsulation changes to implementation might “break” the program and lead to many further changes.
- Why would we want to change the implementation?
  - Different implementations have different tradeoffs (e.g., Space conservation, efficiency etc.).



# Non-API Comments

- Java still uses the `//` and `/* ... */` style comments
- Those comments wouldn't be included in the documentation – and they are used to clarify certain parts of the code

If you have a complicate part in your code,  
don't document it - simplify it !