Creating Data Types : Design Classes

Dr. 何明昕, He Mingxin, Max

program06 @ yeah.net

Email Subject：(AE|A2|A3) + *(Last 4 digits of ID) + Name*: *TOPIC*

Sakai: CS102A in 2018A

计算机程序设计基础

Introduction to Computer Programming

Creating classes

Last time we saw how to use a class:

- create a new object, using `new`;
- send the object messages from its interface, to invoke its behaviour;
- we understood that the object might change its state;
- and that state and behaviour interdepend;
- but we did not expect to have access to the state, and we did not know or need to care exactly how the behaviour was implemented.

This time we will see how to define a class, including its state and behaviour, and how new objects should be created.

Contents

- There are different approaches to writing computer programs.
- They all involve decomposing your programs into parts.
- What is different between the approaches:
  - how the decomposition occurs
  - criteria used for breaking things down
- The approaches to decomposition:
  - Procedural (Structured) Approach
  - Object-Oriented Approach
- Creating Classes and Using Objects

Function-Behaviour-Structure Ontology

https://en.wikipedia.org/wiki/Function-Behaviour-Structure_ontology

Gero J.S. (1990) "Design prototypes: a knowledge representation schema for design", AI Magazine, 11(4), pp. 26–36.

- Function (F): the teleology (purpose) of the design object.
- Behavior (B): the attributes that can be derived from the design object's structure.
- Structure (S): the components of the design object and their relationships.
- The three ontological categories are interconnected:

Function is connected with behavior, and behavior is connected with structure. There is no connection between function and structure!

Conway's Law

Conway's Law:

The rule that the organization of the software and the organization of the software team will be congruent; commonly stated as "If you have four groups working on a compiler, you'll get a 4-pass compiler".

The original statement was more general, "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." This first appeared in the April 1968 issue of *Datamation*.

http://www.catb.org/~esr/jargon/html/C/Conways-Law.html

One of the Reason to explain Why Software Development is difficult:

How to judge whether a design is good or not is a big challenge, that is largely based on experience.

How to Decompose Systems into Modules

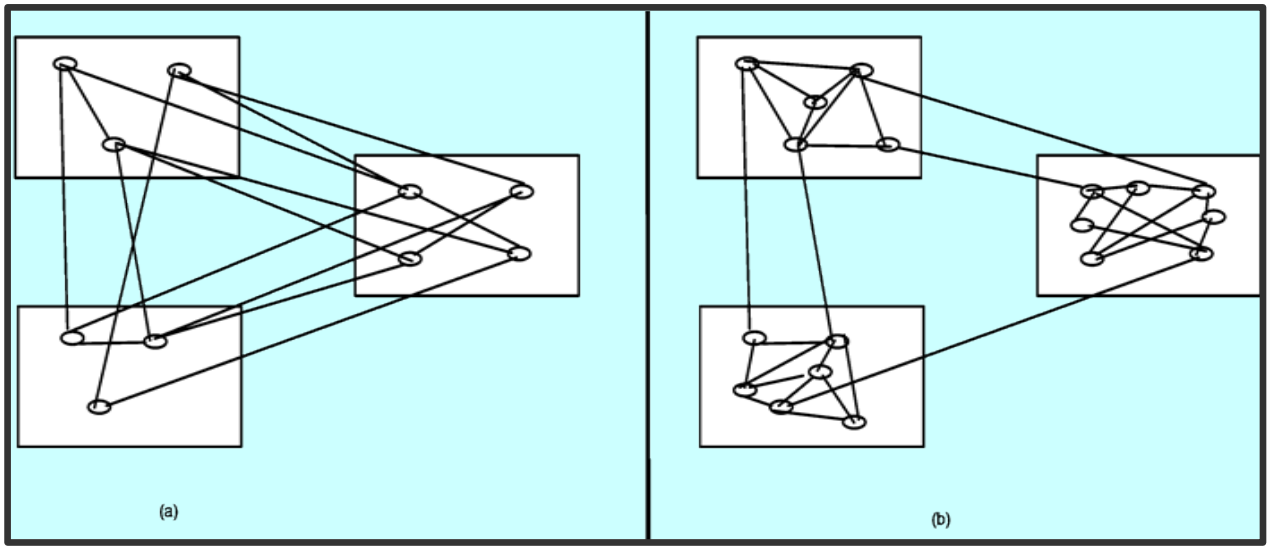
Parnas (1972): SoC, 关注点分离

✓High Cohesion within modules
✓Loose Coupling between modules
✓Information Hiding

*Parnas, D.L. (Dec. 1972). "On the Criteria To Be Used in Decomposing Systems into Modules". Communications of the ACM. **15** (12): 1053–58.*

高内聚，低偶合，信息隐藏
-- 软件开发十字真经

A visual representation



high coupling
(low cohesion)

low coupling
(high cohesion)

Modularity, Cohesion and Coupling

- A complex system may be divided into simpler pieces called *modules*（模块）
- A system that is composed of modules is called *modular*
- Supports application of separation of concerns (SoC, 关注点分离）
  - when dealing with a module we can ignore details of other modules

- Each module should be *highly cohesive*
  - module understandable as a meaningful unit
  - Components of a module are closely related to one another
- Modules should exhibit *low coupling*
  - modules have low interactions with others
  - understandable separately

Structured (process-oriented) Program Design

- A complex system may be divided into simpler pieces called *modules*
- Divide and Conquer 分而治之
- Stepwise Refinement 逐步求精
- 将复杂问题做合理的分解，再分别仔细研究问题的不同侧面（关注点），最后综合各方面的结果，合成整体的解决方案。

Two Papers on this topic by the Instructor:

1) 关注点分离在计算思维和软件工程中的方法论意义,
   计算机科学,2009, 36(4):60-63

2) 面向问题的系统化程序设计方法及其描述工具,
   计算机科学, 1995, 22(5):54-57

3.1 关注点分离的一般模式

对一个典型的复杂问题 P，通过关注点分离解决问题的基本思路可一般地描述为以下 3 个步骤[13]：

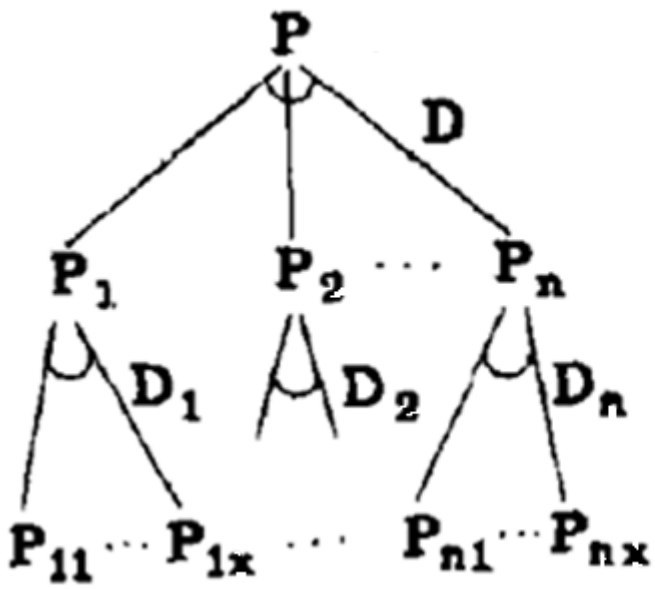
1）先将待解问题 P 分解为不同的关注点 P_1，P_2，…，P_n，即：$P \rightarrow D(P_1, P_2, \cdots, P_n)$，$D$ 表示问题分解策略，即关注点分离方法；

2）对 P_1，P_2，…，P_n 分别考虑，求得各自的解 S_1，S_2，…，S_n；

3）通过对 S_1，S_2，…，S_n 的合成，求得问题 P 的解 S，即：$C(S_1, S_2, \cdots, S_n) \rightarrow S$，$C$ 表示解的合成方法。

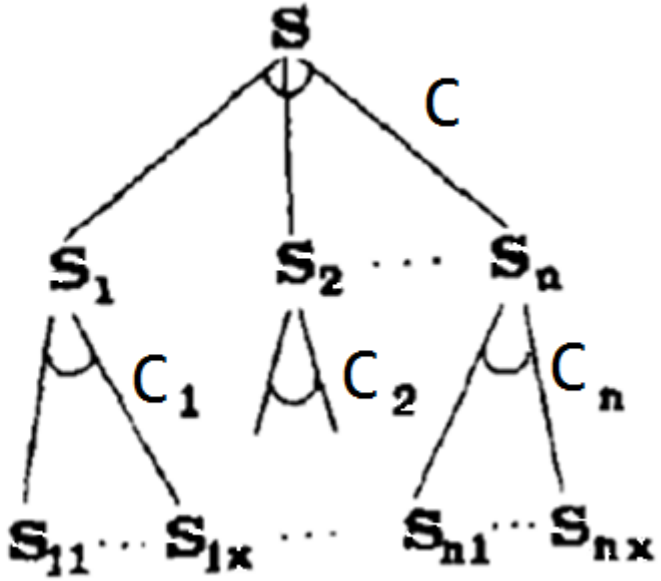
解的合成方法与关注点分离策略密切相关，对具体问题做具体的分析，充分掌握问题相关的具体知识，把握住其关键特征，在关节处实施分割，往往事半功倍。

对复杂问题可能需要多层次多维度的分解。自顶向下逐步分解求精的软件设计方法以及螺旋递增的软件开发过程都是例证。



Problem Decomposition Tree

问题分解树

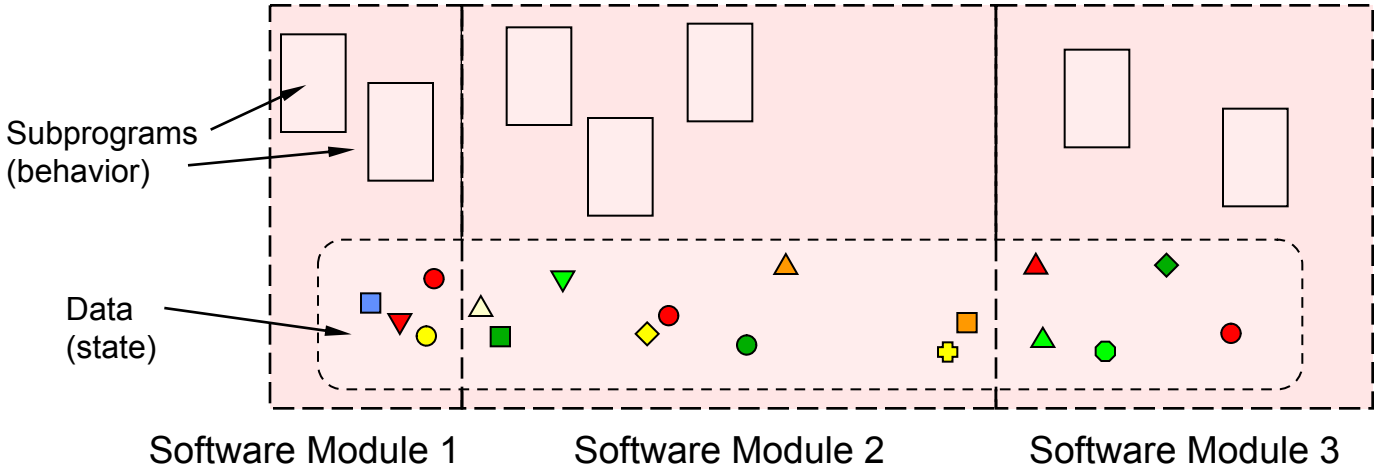


Program Composition Graph

程序生成(构成关系)图

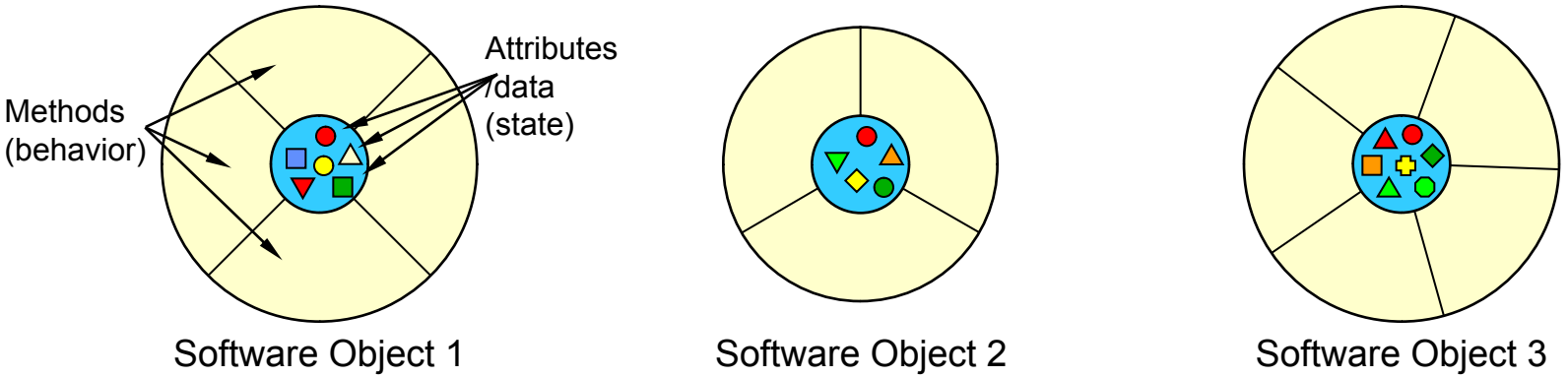
Modules versus Objects

Modules are loose groupings of subprograms and data



"Promiscuous" access to data often results in misuse

Objects ***encapsulate*** data



Object vs. Process-Oriented (1)

- **Process-oriented** is more intuitive because it is individual-centric
  - thinking what to do next, which way to go
- **Object-oriented** may be more confusing because of labor-division
  - Thinking how to break-up the problem into tasks, assign responsibilities, and coordinate the work
  - It's a management problem…

Object vs. Process-Oriented (2)

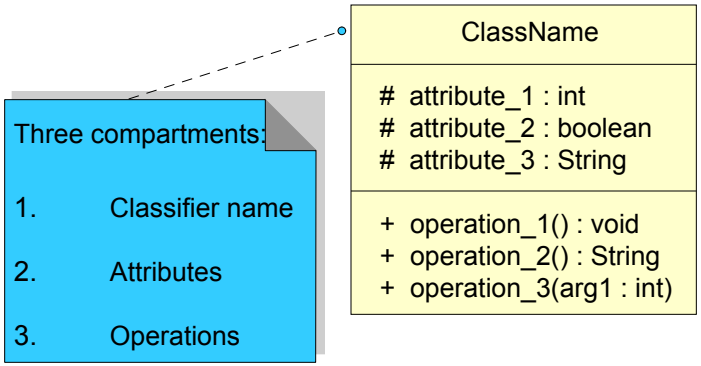
- **Process-oriented** does not scale to complex, large-size problems
  - Individual-centric, but…

Large scale problems require organization of people instead of individuals working alone

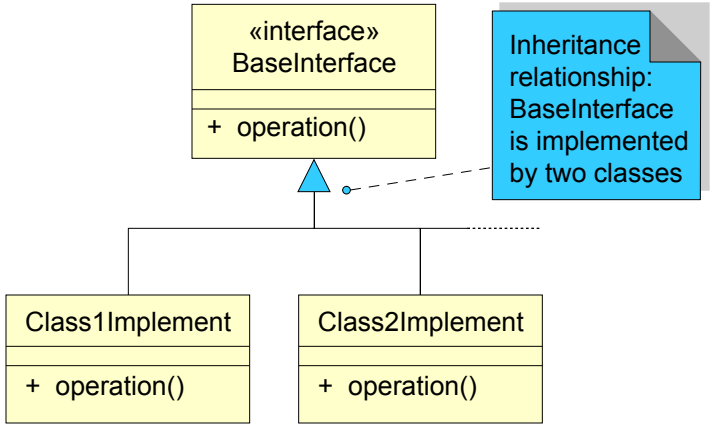
- **Object-oriented** is organization-centric
  - But, hard to design well organizations…

UML Notation for Classes

Software Class



Software Interface Implementation



Classes and Objects

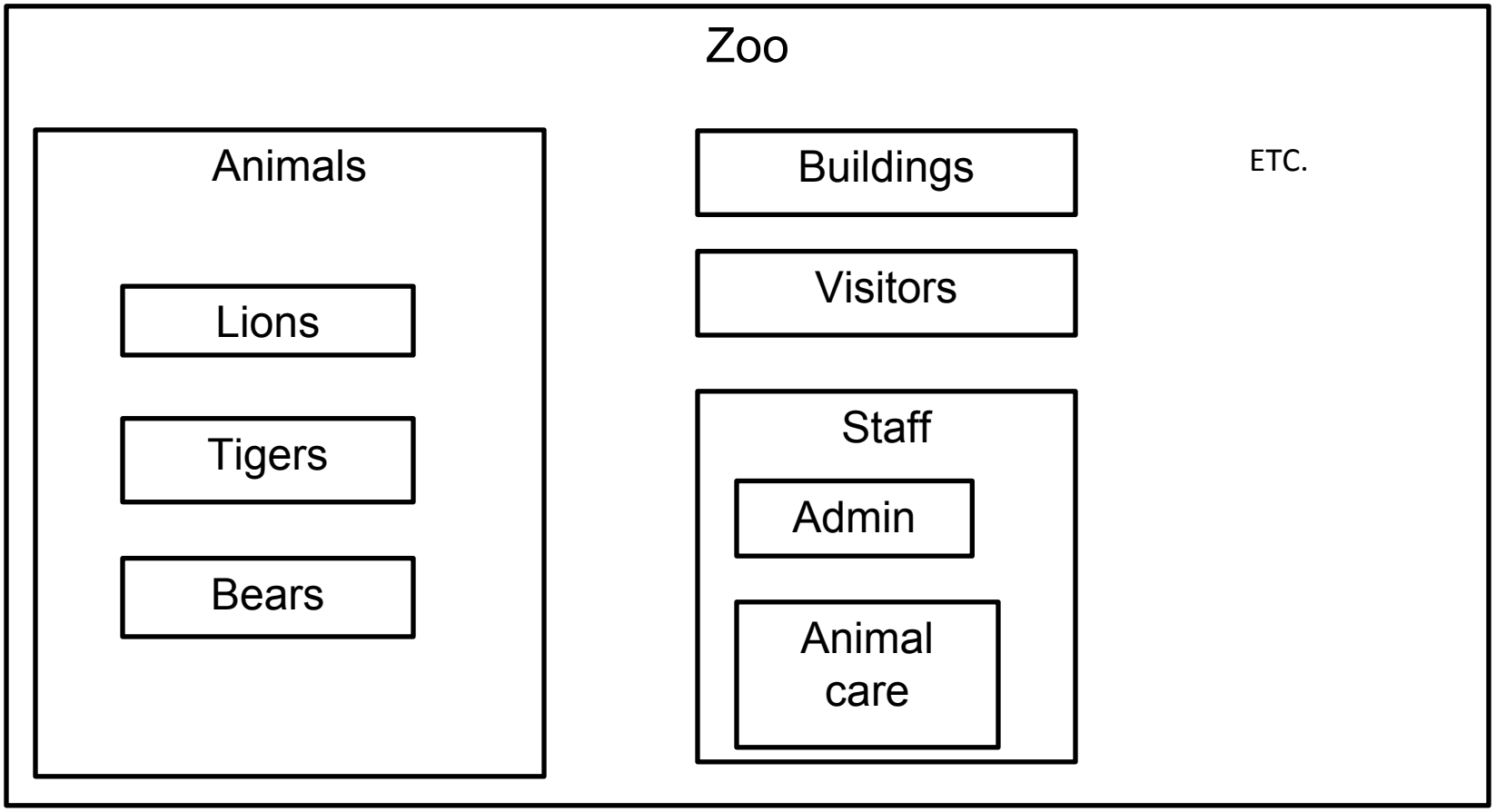
- How to break your program down into objects

  (**Term: "Object-Oriented programming"**)

- This and related topics comprise most of the remainder of the course

An Example Of The Object-Oriented Approach (Simulation)

- Break down the program into entities (classes/objects - described with *nouns*)



Classes/Objects

- Each class of object includes descriptive data.
  - Example (animals):
    - Species
    - Color
    - Length/height
    - Weight

    etc.
- Also each class of object has an associated set of actions
  - Example (animals):
    - Sleep
    - Eat
    - Excrete

    etc.

Types In Computer Programs

- Programming languages typically come with a built in set of types that are known to the translator

```
int num;
// 32 bit integer (e.g. operations: +, -, *, /, %)

String s = "Hello";
// Unicode character information (e.g. operation: concatenation)
```

- Unknown types of variables cannot be arbitrarily declared!

```
Person max;
// What info should be tracked for a Person
// What actions is a Person capable of
// Compiler error!
```

A Class Must Be First Defined

- A class is a new type of variable.
- The class definition specifies:
  - What descriptive data is needed (values, 值)?
    - Programming terminology: attributes (属性) = data (**New definition**)
  - What are the possible set of actions (operations, 操作)?
    - Programming terminology: methods (方法) = actions (**new definition**)
    - A method is the Object-Oriented equivalent of a function/procedure (函数/过程)

RECAP *Type*: A set of values together with operations on them and a limited storage with specific format.

Defining A Java Class

Format:

```
public class <className> {
    attributes
    constructors
    methods
}
```

Example (more explanations coming shortly):

```
public class Person {
    private int age;                    // Attribute

    public Person (int anAge) {         // Constructor
        age = anAge;
    }

    public String sayAge () {           // Method
        Return “My age is " + age;
    }
}
```

Simple Object-Oriented Example

- Program design: each class definition

  e.g., `public class <className>`

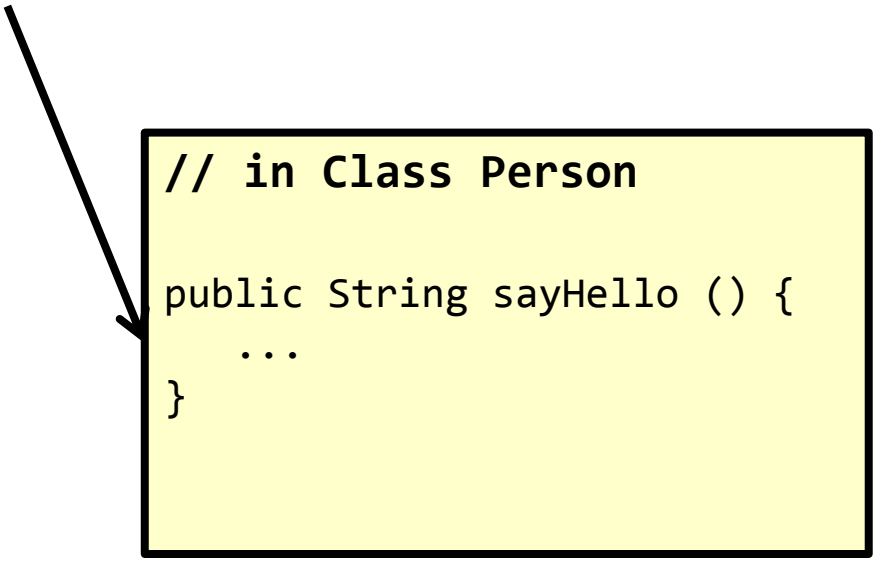
  must occur its own "dot-java" file, `<className>.java`

- Example program consists of two files in the same directory:
  - (From now on your programs must be laid out in a similar fashion):

    `Driver.java`
    `Person.java`

The Driver Class (Client)

```
public class Driver {
   public static void main (String [] args) {
      Person aPerson = new Person();
      System.out.println( aPerson.sayHello() );
   }
}
```



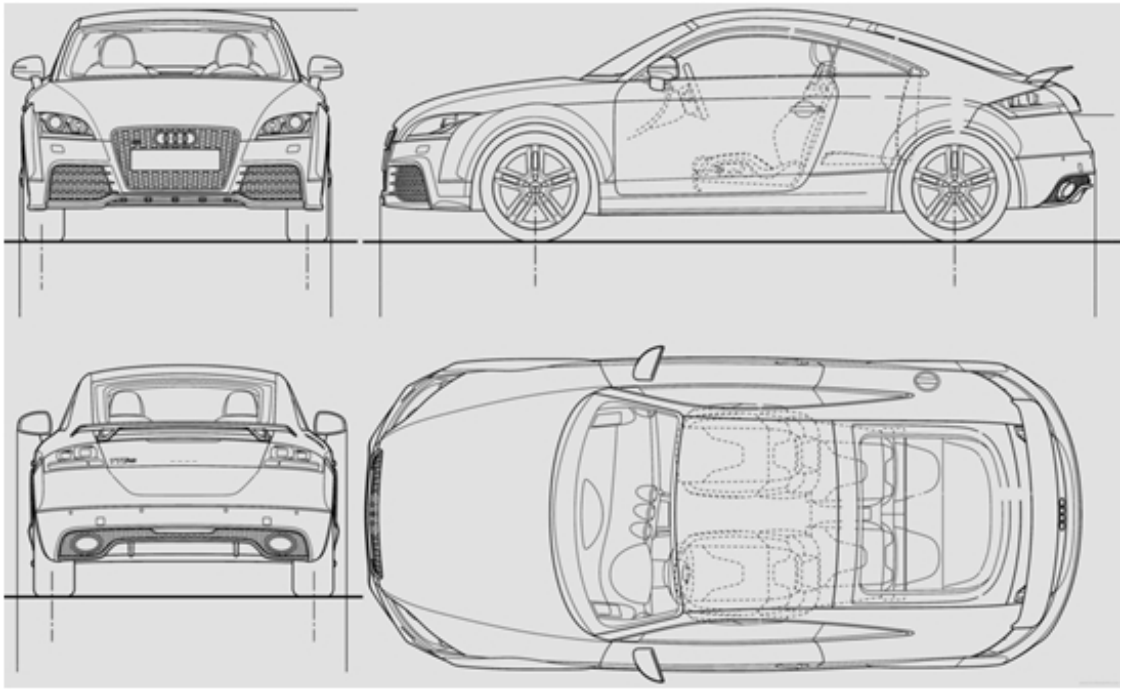

```
I don't wanna say hello.
```

Class Person

```
public class Person {
    …
    public String sayHello () {
       return "I don't wanna say hello.”;
    }
}
```

Concepts: Classes Vs. Objects

- Class:
  - Specifies the characteristics of an entity but is not an instance of that entity
  - Much **like a blue print** that specifies the characteristics of a car (shape, height, width, length, weight, color etc.)



Concepts: Classes Vs. Objects (2)

- Object:
  - A specific example or instance of a class.
  - Objects have all the attributes specified in the class definition

Special Method: main(String[])

- Language requirement: There must be a `main()` method - or equivalent – to determine the starting execution point.
- Syntactic Format:

```
public static void main (String[] args) {
}
```

- `String[] args` are Command-Line Arguments
- Style requirement: the name of the class that contains `main()` is often referred to as the *<Driver>* class.
  - Makes it easy to identify the starting execution point in a big program.
- Do not instantiate instances of the *<Driver>*
- Normally, Avoid:
  - Defining attributes for the `Driver`
  - Defining methods for the `Driver` (other than the `main()` method)

Compiling Multiple Classes

- One way (safest) is to compile all code (dot-Java) files when any code changes.
- Example:

```
-> javac Driver.java
-> javac Person.java
```

or

```
-> javac Driver.java Person.java
```

Alternatively use the 'wildcard' :

```
-> javac *.java
```

```
-> javac Driver.java
```

Will automatically compile Person.java in the same folder, due to that class Driver used class Person!

Why Must Classes Be Defined

- Some classes are already pre-defined (included) in a programming language with a list of attributes and methods e.g., `String`, PrintStream
- Why don't more classes come 'built' into the language?
- The needs of the program will dictate what attributes and methods are needed.

Defining The Attributes Of A Class In Java

- Attributes can be variable or constant (preceded by the '`final`' keyword), for now stick to the former.
- **Format**:

  <accessModifier>[1] *<attributeType>* *<attributeName>*;

- **Example**:

```
public class Person {
   private int age;
}
```

1) Although other options may be possible, *attributes are almost always set to private* (more on this later).

Term: Object State

- Similar to how two variables can contain different data.
- Attributes: Data that describes each instance or example of a class.
- Different objects have the same attributes but the values of those attributes can vary
  - Reminder: The class definition specifies the attributes and methods for *all objects*
- Example: two 'monster' objects each have a health attribute but the current value of their health can differ
- The current value of an object's attribute's determines it's state.

```
Age: 35
Weight: 192
```

```
Age: 50
Weight: 125
```


```
Age: 1.5
Weight: 15
```

Defining The Methods Of A Class In Java

Format:

```
<accessModifiers>[1] <returnType[2]> <methodName> (<p1Type> <p1Name>, <p2Type> <p2Name>…) {
    <Body of the method>
}
```

Example:

```
public class Person {
    // Method definition
    public String sayAge () {
       return “My age is " + age;
    }
}
```

1) For now *set the access modifier on all your methods to ‘public’* (more on this later).

2) Return types: includes all the built-in ‘simple’ types such as char, int, double…arrays and classes that have already been defined (as part of Java or third party extras)

Parameter Passing: Different Types

Parameter Type	Format	Example
Simple Types	`<method>(<type> <name>)`	`method(int x, char y) { ... }`
Objects	`<method>(<class> <name>)`	`method(Person p) { ... }`
Arrays	`<method>(<type>[]… <name>)`	`method(double[][] m) { ... }`

When calling a method, only the values of the parameters must be passed.

```
System.out.println( num );
```

Return Values: Different Types

Return Type	Format	Example
Simple Types	`<type> <method>()`	`int method() { return 0; }`

Objects	`<class> <method>()`	`Person method() {`
		`  Person p = new Person();`
		`  return p;`
		`}`

Arrays	`<type>[]... <method>()`	`Person[] method() {`
		`  Person[] team = new Person[3];`
		`  return team;`
		`}`

What Are Methods

- Possible behaviors or actions for each instance (example) of a class.

walk()
talk()

walk()
talk()

fly()

swim()

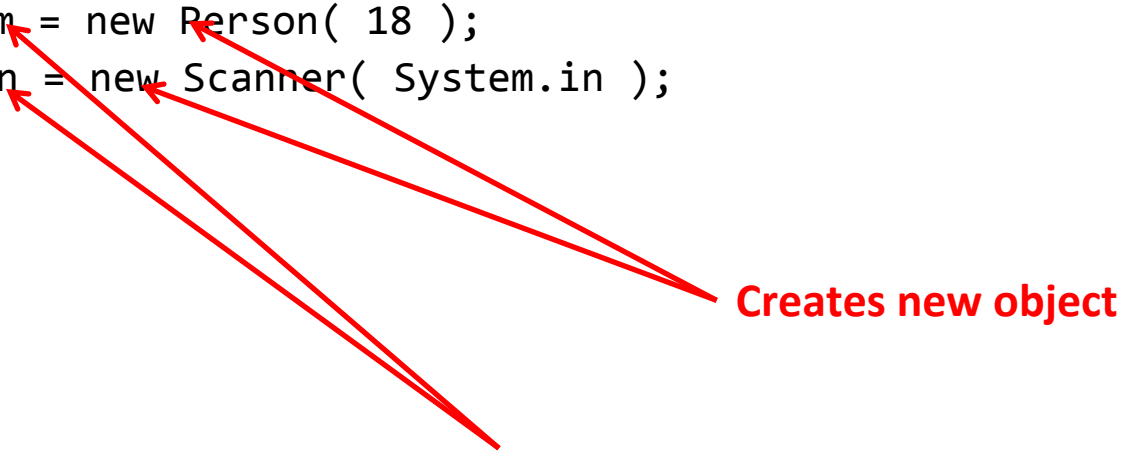
Instantiation

- **Definition**: Instantiation, creating a new instance or example of a class.
- Instances of a class are referred to as *objects*.
- **Format**:

```
<className> <instanceName> = new <className>(…);
```

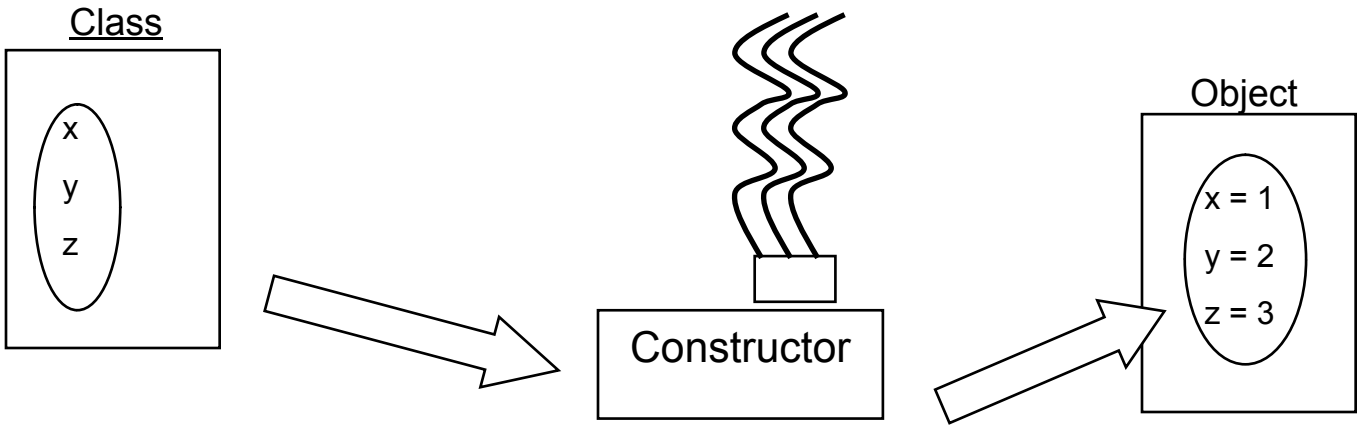
- **Examples**:

```
Person jim = new Person( 18 );
Scanner in = new Scanner( System.in );
```



Constructor

- **Term**: A special method to initialize the attributes of an object as the objects are instantiated (created).



- The constructor is automatically invoked whenever an instance of the class is created e.g., `Person aPerson = new Person();`

Call to constructor (creates something 'new')

```
class Person {
    // Constructor
    public Person () {
        …
    }
}
```

- Constructors can take parameters but **never** have a return type.

Term: Default Constructor

- Takes no parameters
- If no constructors are defined for a class then a default constructor comes 'built-into' the Java language.
- e.g.,

```
.. class Driver {
    .. main (..) {
        Person aPerson = new Person();
    }
}

.. class Person {
    private int age;
}
```

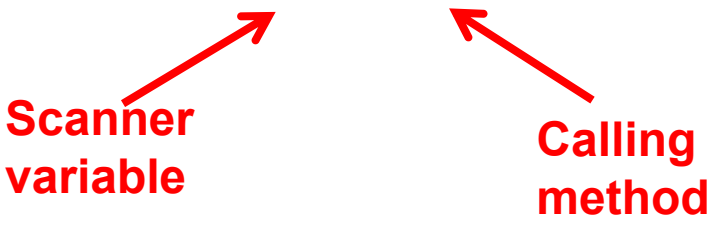
Calling Methods (Outside The Class)

- You've already done this before with pre-created classes!
- First create an object (previous slides)
- Then call the method for a particular variable.
- **Format**:

  <instanceName>.<methodName>(<p1Name>, <p2Name>..);

- **Examples:**

```
Person jim = new Person();
String s = jim.sayHello();

// Example, calling Scanner class method
Scanner in = new Scanner( System.in );
System.out.print( "Enter your age: " );
age = in.nextInt();
```



Calling Methods: Outside The Class You've Defined

- Calling a method outside the body of the class (i.e., in another class definition)
- The method must be prefaced by a variable (actually a reference to an object – more on this later).

```
public class Driver {
   public static void main (String [] args) {
      Person bart = new Person();
      Person lisa = new Person();

      becomeOlder();        // Incorrect! Who ages?

      bart.becomeOlder();   // Right. Happy birthday Bart!
    }
}
```

Calling Methods: Inside The Class

- Calling a method inside the body of the class (where the method has been defined)
  - You can just directly refer to the method (or attribute)

```
public class Person {
  private int age;
  ..

  public void birthday() {
     becomeOlder();  // access a method
  }

   public void becomeOlder() {
     age++;     // access an attribute
   }
}
```

Another Object-Oriented Example

- Learning concepts:
  - Attributes
  - Constructors
  - Accessing class attributes in a class method

Class Driver

```
public class Driver {
   public static void main (String [] args) {
      Scanner in = new Scanner( System.in );
      System.out.print( "Enter age: " );
      int age = in.nextInt();
      Person jim = new Person( age );
      System.out.println(  jim.sayAge() );
    }
}
```

```
// Class Person
public Person (int anAge) {
   age = anAge;
}
```

```
// Class Person
public String sayAge() {
   return "My age is " + age;
}
```

```
> java Driver
Enter age: 123
My age is 123
```

```
> java Driver
Enter age: 321
My age is 321
```

Class Person

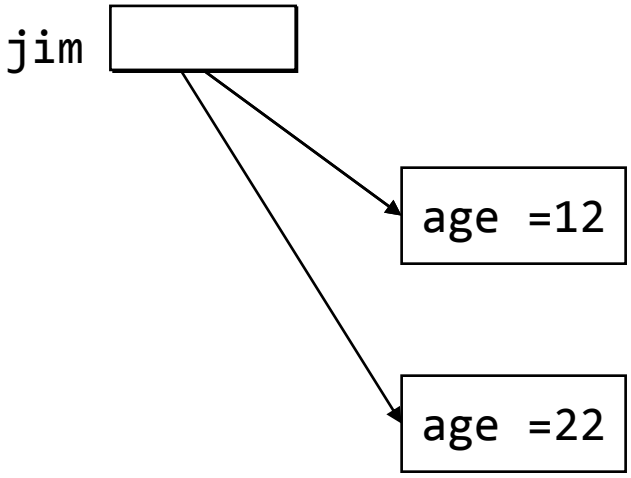
```
public class Person {
   private int age;

   public Person (int anAge) {
      age = anAge;
   }

   public String sayAge () {
      return "My age is " + age;
   }

   public void birthday() {
      becomeOlder();
   }

   public void becomeOlder() {
      age++;
   }
}
```

Creating An Object

- Two stages (can be combined but don't forget a step)
  - Create a variable that refers to an object e.g., `Person jim;`
  - Create a *new* object e.g., `jim = new Person(12);`
    - The keyword 'new' calls the constructor to create a new object in memory
  - Observe the following

Jim is a reference to a Person object

```
Person jim;

jim = new Person(12);

jim = new Person(22);
```



Terminology: Methods Vs. Functions

- Both include defining a block of code that be invoked via the name of the method or function (e.g., `print()`)
- **Methods** a block of code that is *defined within a class definition* (Java example):

```
public class Person {
    public Person (int age) { ... }

    public String sayAge () { ... }
}
```

- Every object that is an instance of this class (e.g., `jim` is an instance of a `Person`) will be able to invoke these methods.

```
Person jim = new Person(18);
..println( jim.sayAge() );
```

Methods Vs. Functions (Static methods): Summary

Methods

- The Object-Oriented approach to program decomposition.
- Break the program down into classes.
- Each class will have a number of methods.
- Methods are invoked/called through an instance of a class (an object).

Functions

- The procedural (procedure = function) approach to program decomposition.
- Break the program down into functions.
- Functions can be invoked or called without creating any objects.

Second Example: Second Look

Calls in Driver.java

```
Person jim = new Person(8);

..( jim.sayAge() ) ;
```

Person.java

```
public class Person {
    private int age;

    public Person (int anAge) {
        age = anAge;
    }

    public String sayAge () {
        return "My age is " + age;
    }
    ..
}
```

More is needed:

- What if the attribute 'age' needs to be modified later?
- How can age be accessed but not just via a `print()`?

Viewing And Modifying Attributes

1) Terms: Accessor methods: 'get()' method

- Used to determine the current value of an attribute
- Example:

```
public int getAge() {
    return age;
}
```

2) Terms: Mutator methods: 'set()' method

- Used to change an attribute (set it to a new value)
- Example:

```
public void setAge (int anAge) {
    age = anAge;
}
```

Class Person

- Notable differences: constructor is redesigned, `getAge()` replaces `sayAge()`, `setAge()` method added

```
public class Person {
   private int age;

   public Person () { age = 0; }

   public Person (int anAge) { age = anAge; }

   public int getAge () { return age; }

   public void setAge (int anAge)  { age = anAge; }

   public String sayAge () { return "My age is " + age; }

   public void birthday () { becomeOlder(); }

   public void becomeOlder () { age++; }
}
```

Class Driver

```
public class Driver {
    public static void main (String [] args) {
        Person jim = new Person();
        System.out.println( jim.getAge() );
        jim.setAge( 21 );
        System.out.println( jim.getAge() );
    }
}
```

Constructors

- Constructors are used to initialize objects (set the attributes) as they are created.
- Different versions of the constructor can be implemented with different initializations e.g., one version sets all attributes to default values while another version sets some attributes to the value of parameters.
- **New term:** method overloading, same method name, different parameter list.

```
public Person (int anAge) {
   age = anAge;
   name = "No-name";
}
```

```
public Person () {
   age = 0;
   name = "No-name";
}
```

```
// Calling the versions (distinguished by parameter list)
Person p1 = new Person(100);    Person p2 = new Person();
```

Class Person

```
public class Person {
   private int age;
   private String name;

   public Person ()                          { age = 0;     name = "No-name"; }
   public Person (int anAge)                 { age = anAge; name = "No-name"; }
   public Person (String aName)              { age = 0;     name = aName; }
   public Person (int anAge, String aName) { age = anAge; name = aName; }

   public int getAge ()             { return age; }
   public void setAge (int anAge)  { age = anAge; }

   public String getName ()             { return name; }
   public void setName (String aName)  { name = aName; }

   public String sayAge ()    { return "My age is " + age; }

   public void birthday ()    { becomeOlder(); }

   public void becomeOlder () { age++; }
}
```

Class Driver

```
Person()
Person(int)
Person(String)
Person(int,String)
```

```
public class Driver {
   public static void main (String [] args) {
      Person jim1 = new Person();              // age, name default
      Person jim2 = new Person( 21 );          // age=21
      Person jim3 = new Person( "jim3" );      // name="jim3"
      Person jim4 = new Person( 65, "jim4" ); // age=65, name="jim4"


      System.out.println( jim1.getAge() + " " + jim1.getName() );
      System.out.println( jim2.getAge() + " " + jim2.getName() );
      System.out.println( jim3.getAge() + " " + jim3.getName() );
      System.out.println( jim4.getAge() + " " + jim4.getName() );
   }
}
```

```
0 No-name
21 No-name
0 jim3
65 jim4
```

Terminology: Method Signature

- Method signatures consist of: the method name, the type, number and order of the parameters.
- The signature will determine the overloaded method called:

```
Person p1 = new Person();
Person p2 = new Person( 25 );
```

Overloading(重载) And Good Design

- Overloading: methods that implement similar but not identical tasks.
- Examples include class constructors but this is not the only type of overloaded methods:

```
System.out.println( int )
System.out.println( double )
```

  etc.

  For more details on class System on Java API Documentation.

- Benefit: just call the method with required parameters.

Method Overloading: Things To Avoid

- Distinguishing methods solely by the order of the parameters.

```
m(int,char);
```

Vs.

```
m(char,int);
```

- Overloading methods but having an identical implementation.
- Why are these things bad?

Method Signatures And Program Design

- Unless there is a compelling reason do not change the signature of your methods!

Before:

```
.. class Foo {
   .. void fun() {

   }
}
```

After:

```
.. class Foo {
   .. void fun (int num) {

   }
}
```

```
public static void main (..) {
   Foo f = new Foo();
   f.fun();
}
```

This change has broken me! ☹

Back to the 'private' Keyword

- It syntactically means this part of the class cannot be accessed outside of the class definition.
  - You should **always** do this for variable attributes, *very rarely do this* for methods (more later).
- Example

```
public class Person {
   private int age;
   public Person() {
      age = 12;   // OK – access allowed here
   }
}
public class Driver {
   public static void main (String [] args) {
      Person aPerson = new Person();
      aPerson.age = 12;  // Syntax error: program won't compile!
   }
}
```

Classes and Clients

Client code:

- In general, a client program calls a method of some class `C`.
- Example: class `FooTester` is a client of `Foo` because it calls the `doSomething()` instance method on `Foo` objects.

Test-first design methodology:

1. Think about the methods a client would call on instances of class `C`.
2. Design the API for class `C`.
3. Implement a client `CTester` for `C` which tests the desired behaviour.
4. Implement `C` so that it satisfies `CTester`.

CircleTester

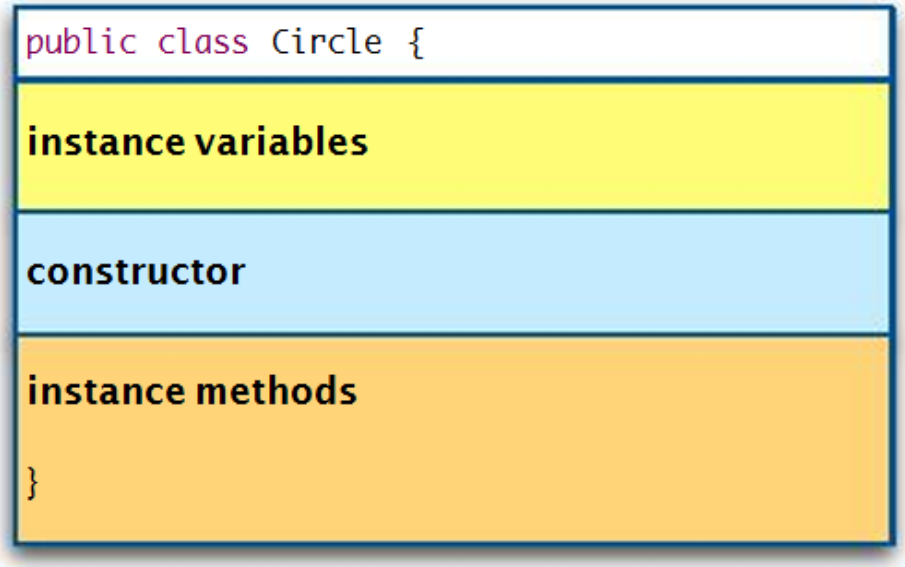
- Create a `Circle` object `c1`.
- Call a method to get the area of that object: `c1.getArea()`

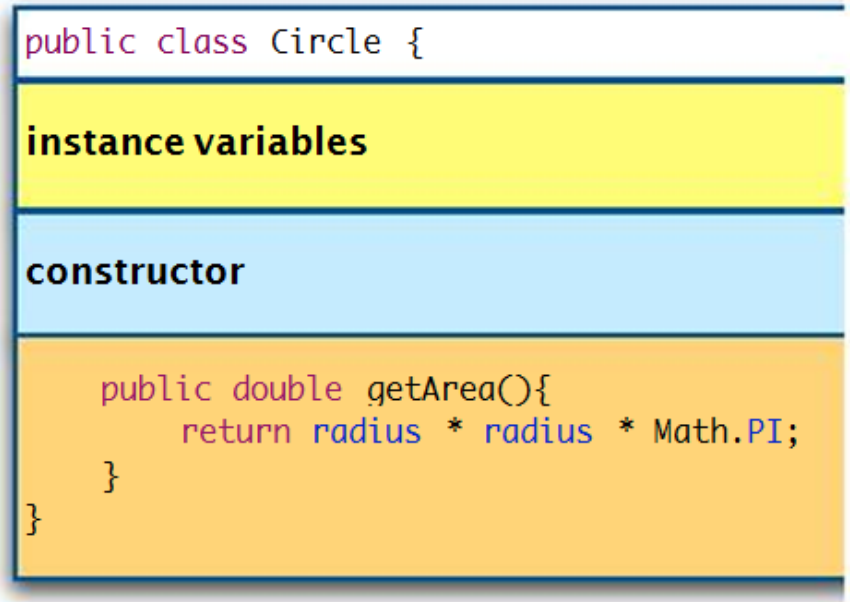
```
public class CircleTester {

    public static void main(String[] args) {
        Circle c1 = new Circle();
        double area1 = c1.getArea();
        System.out.printf("Area of circle c1 is %5.2f\n", area1);

        Circle c2 = new Circle(5.0);
        double area2 = c2.getArea();
        System.out.printf("Area of circle c2 is %5.2f\n", area2);
    }
}
```

Expected Output

```
% java CircleTester
Area of circle c1 is  3.14
Area of circle c2 is 78.54
```

The `Circle` Class: Instance Methods





- `getArea()` is an instance method of the class `Circle`.
- How does it know about `radius`?

The `Circle` Class: Instance Variables

```
public class Circle {

    private double radius;

constructor

    public double getArea(){
        return radius * radius * Math.PI;
    }
}
```

- `radius` is an instance variable of the class `Circle`.
- Instance variables are declared outside methods and have scope over the whole class.
- An instance method of a class can use any instance variable of that class.
- Instance variables do not have to be initialized; they get default values (e.g., 0 for int, `false` for boolean, `null` for all reference types).
- How does a `Circle` object's radius get set?

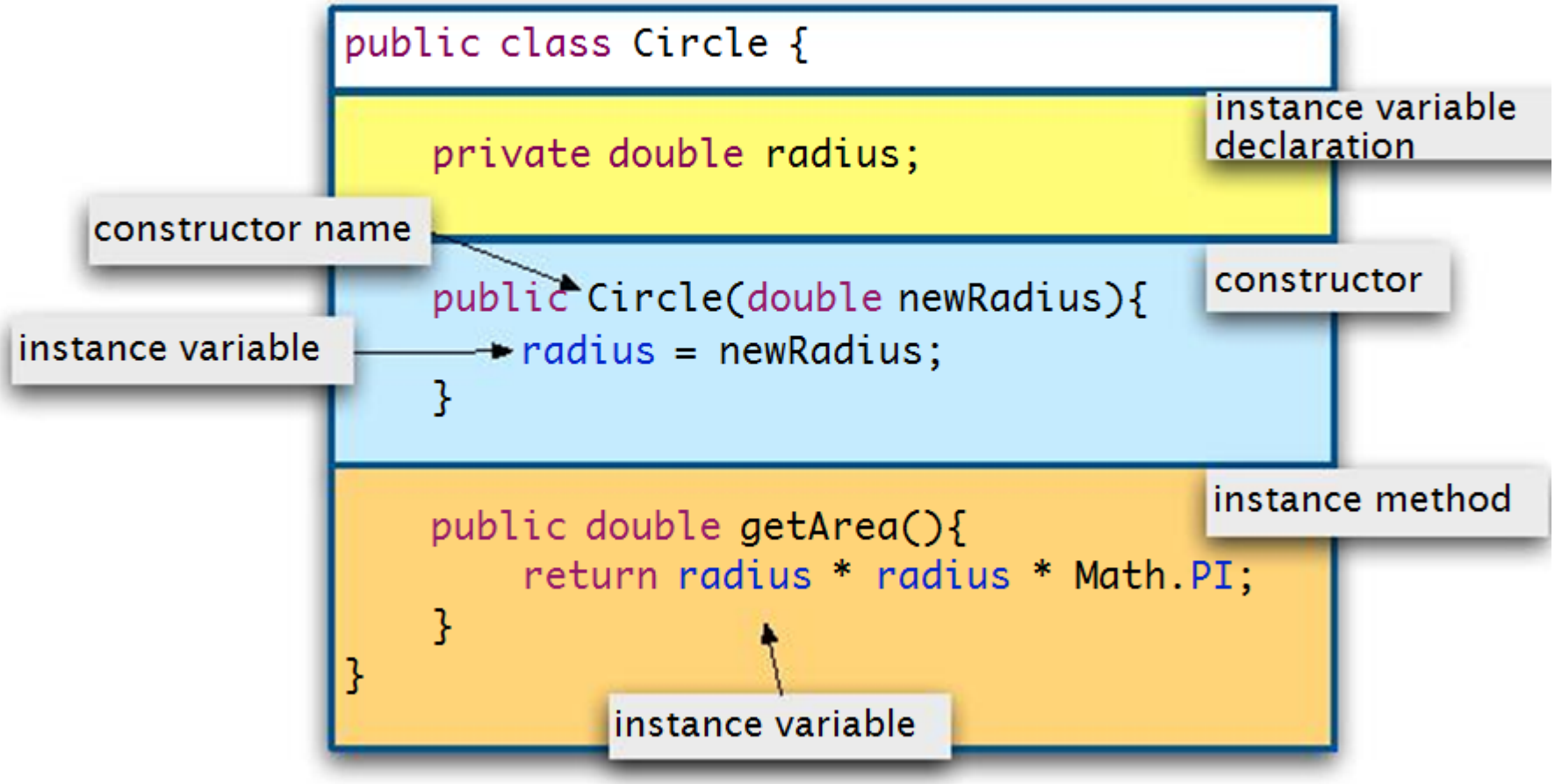
The `Circle` Class: Constructors

```
public class Circle {

    private double radius;

    public Circle(double newRadius){
        radius = newRadius;
    }

    public double getArea(){
        return radius * radius * Math.PI;
    }
}
```

Constructor

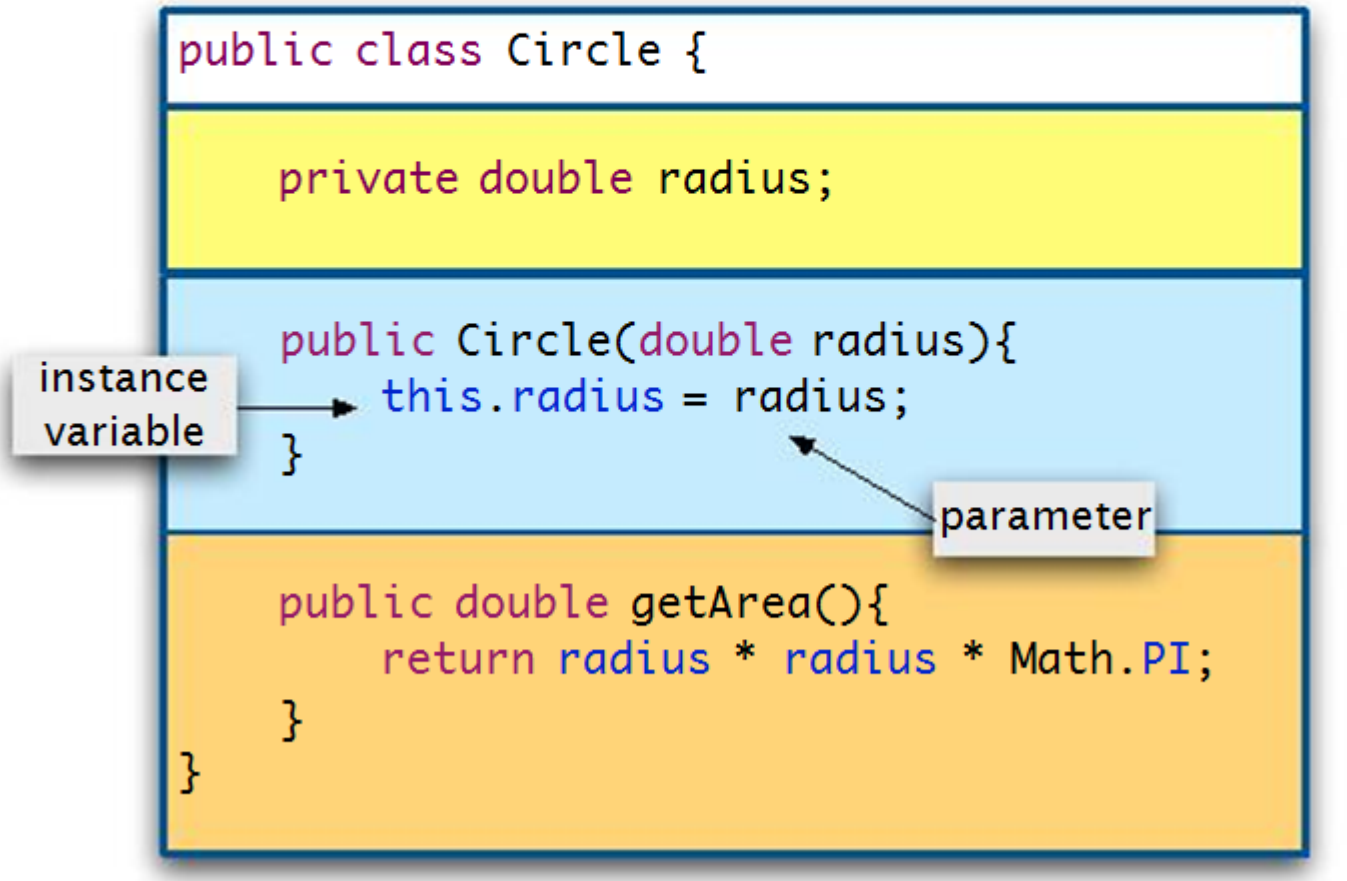
- has same name as the class;
- used to initialize an object that has been created: `new Circle(5.0);`
- must not have a return type (not even `void`).

The Circle Class: Anatomy

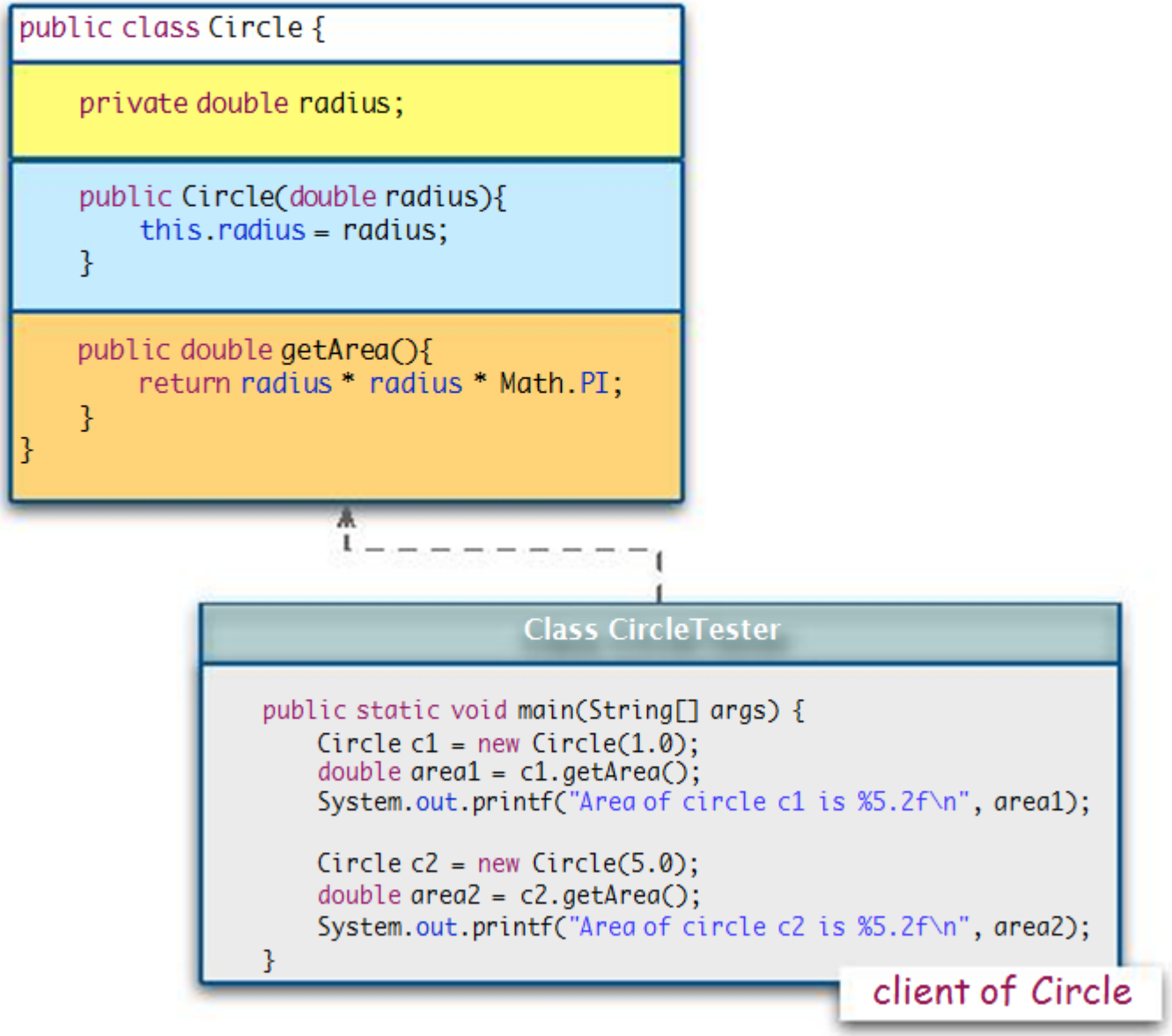


The Circle Class: Constructors

Alternative notation:



The Circle Class: Client



More on Constructors

Circle1: Omitting the constructor

```
public class Circle1 {
    private double radius;
    public double getArea(){
        return radius * radius * Math.PI;
     }
}
```

- `Circle1 c = new Circle1(1.0)` — causes compile-time error.
- `Circle1 c = new Circle1()` — does work
  - though `c.getArea()` returns `0.00`!
- If you don't explicitly add a constructor, Java will automatically add a no-argument constructor for you.

More on Constructors

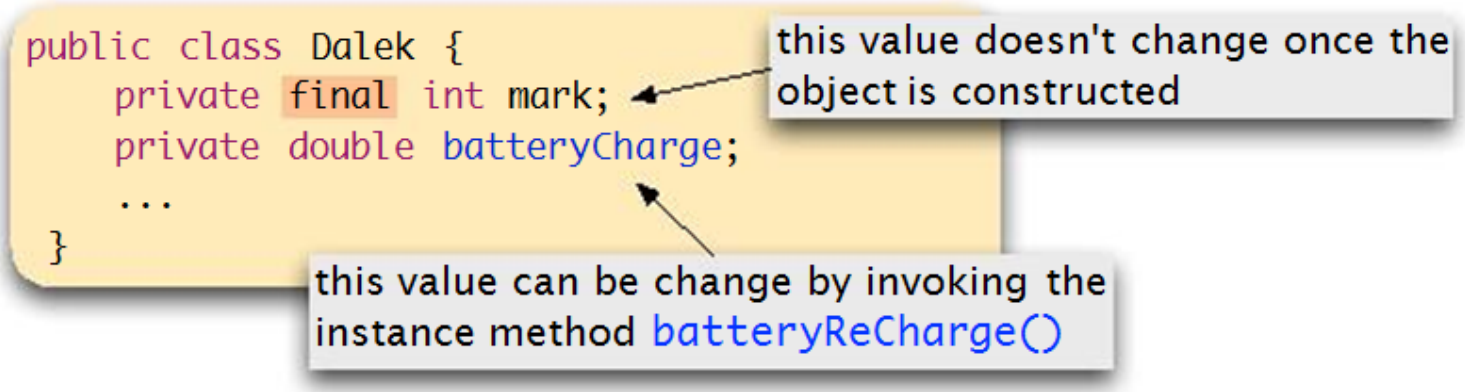
Circle again

```
public class Circle {
    private double radius;
     public Circle(double newRadius){
          radius = newRadius;
     }
    public double getArea(){
          return radius * radius * Math.PI;
     }
}
```

- What happens if we call `Circle c = new Circle()`?
- This also causes a compile-time error — we only get the no-arg default constructor if there's no explicit constructor already defined.

The `final` Modifier

Final: declaring a variable to by `final` means that you can assign it a value only once, in initializer or constructor. E.g., Daleks come in three versions, Mark I, Mark II and Mark III.



Advantages:

- Helps enforce immutability.
- Prevents accidental changes.
- Makes program easier to debug.
- Documents the fact that value cannot change.

Getters and Setters

Encapsulation: instance variables should be private

```
public class Student {

    private String firstName;
    private String lastName;
    private String matric;

    public Student(String fn, String ln, String m) {
        firstName = fn;
        lastName = ln;
        matric = m;
    }
}
```

Getters and Setters

Encapsulation: instance variables should be private

```
public class StudentTester {

    public static void main(String[] args) {
        Student student = new Student("Fiona", "McCleod", "s01234567");
        System.out.println(student.firstName);
        student.matric = "s13141516";
    }
}
```

we cannot assign to this variable!

we cannot read this variable!

Getters and Setters

Encapsulation: instance variables should be private

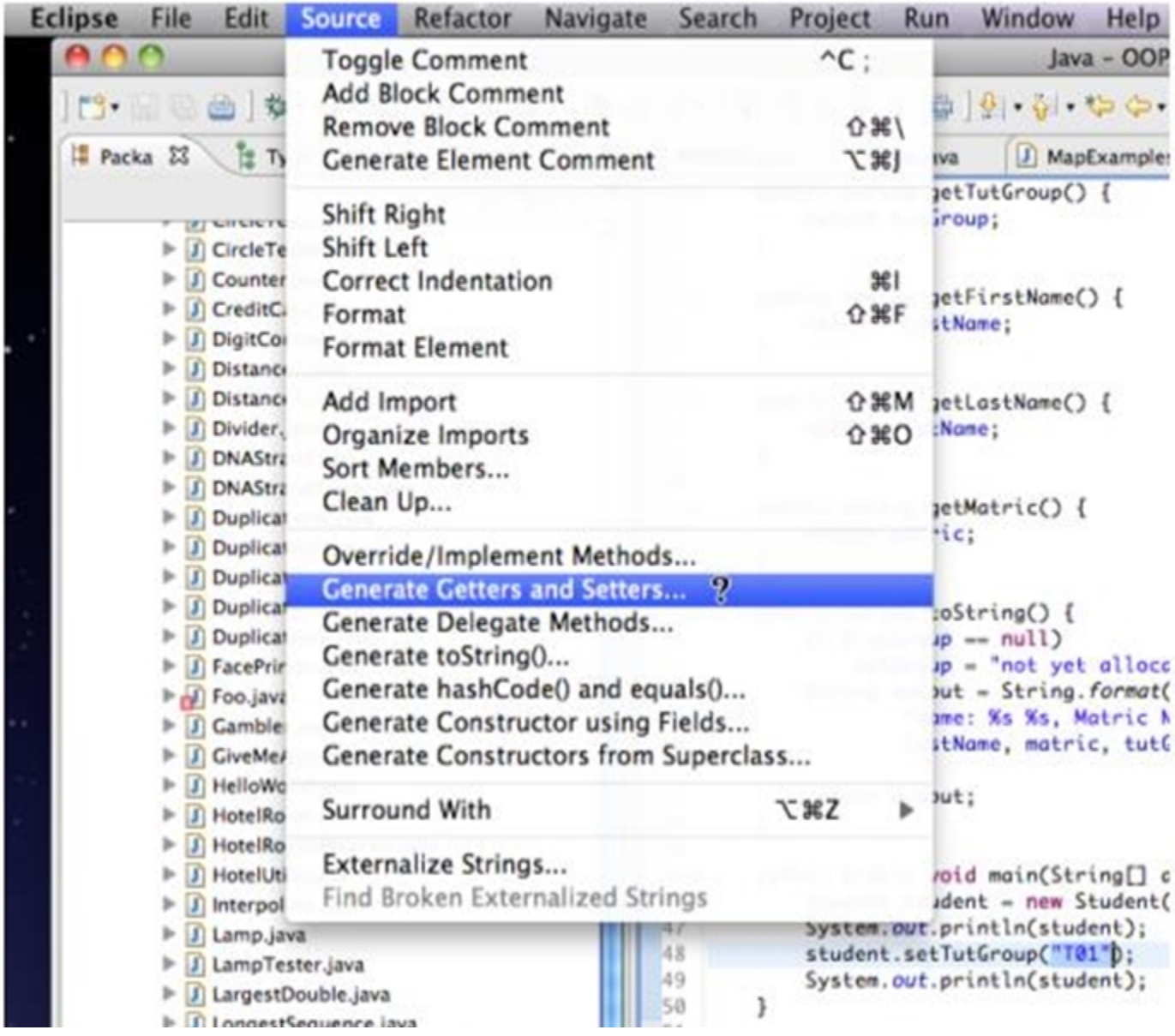
- We use instance methods to mediate access to the data in private instance variables, as needed.
- Accessor methods: just read the data
- Mutator methods: modify the data
- Java convention: given an instance variable `myData`, use
  - `getMyData()` method to read the data, and
  - `setMyData()` method to write to the data.
- Often called 'getters' and 'setters' respectively.

Getters and Setters

```
public class Student {

    private String firstName, lastName, matric, tutGroup;

    public Student(String fn, String ln, String m) {
        ...
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getMatric() {
        return matric;
    }
}
```

Getters and Setters

Eclipse will generate setters and getters for you!



Summary: Object Orientation

Data type: set of values and collections of operations on those values.

In OOP: classes.
Simulating the physical world

- Java objects can be used to model real-world objects
- Not necessarily easy to choose good modelling primitives, or to get model that reflects relevant parts of reality.
- Examples: geometric figures, hotel rooms, ...

Extending the Java language

- Java doesn't have a data type for every possible application.
- User-defined classes enable us to add our own abstractions.

Summary: designing a Java class

- Use client code to motivate and test classes.
- instance variables:
  - represent data that is particular to an object (i.e., an instance!);
  - have scope over the whole class;
  - can hold mutable state;
  - can be manipulated by any instance method in the class.
- instance methods:
  - like static methods, but can only be called on some object `o`;
  - have access to the data that is specific to `o`.
- constructors:
  - we create a new object of class `Foo` with the keyword `new`;
  - we initialize an object of type `Foo` by calling the constructor for that type;
  - the constructor can be used to store data values in the object's instance variables.

Summary: Access Control

Encapsulation and visibility: All the instance variables and methods (i.e., members) of a class are visible within the body of the class.

Access modifiers: control the visibility of your code to other programs.

`public`: member is accessible whenever the class is accessible.

`private`: member is only accessible within the class.

default: member is only accessible within package.

`protected`: member is only accessible within package and extended classes.

Benefits of encapsulation:

- Loose coupling
- Protected variation
- Exporting an API:
  - the classes, members etc, by which some program is accessed
  - any client program can use the API
  - the author is committed to supporting the API