zOOPs Concepts in Java

Object-oriented programming System(OOPs) is a programming concept that is based on “objects”. The primary purpose of object-oriented programming is to increase the readability, flexibility and maintainability of programs.

Object oriented programming brings data and its behaviour together in a single entity called objects. It makes the programming easier to understand.

- Object
- Class
- Abstraction
- Encapsulation and Data Hiding
- Inheritance
- Polymorphism
- Modularty
- Dynamic Initialization
- Message Passing

computer soccer games

- **Player:** attributes include name, number, x and y location in the field, and etc; operations include run, jump, kick-the-ball, and etc.
- **Ball:** attributes include x, y, z position in the field, radius, weight, etc.
- **Referee:**
- **Field:**
- **Audience:**
- **Weather:**

What is Class

A class can be considered as a **blueprint** which **you can use to create as many objects as you like**.

a *class* is a blueprint, template, or prototype that defines and describes the *static attributes* and *dynamic behaviors* common to all objects of the same kind.

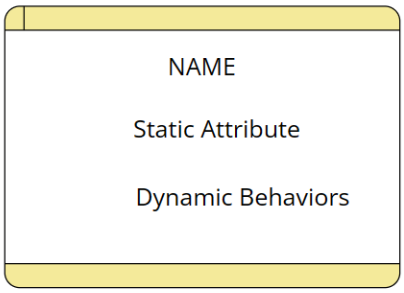
instance

An *instance* is *a realization of a particular item of a class*. In other words, an instance is an *instantiation* of a class. All the instances of a class have similar properties, as described in the class definition.

he term "*object*" usually refers to *instance*.

```
public class Car {
  private String doors;
  private String engine;
  private String driver;
  private int speed;

  public void setSpeed(int speed){
    this.speed = speed;
  }
  public int getSpeed(){
    return speed;
  }

  public static void main(String[] args) {
    Car Bmw=new Car();
    Car Mercedes=new Car();
    Car Volvo=new Car();
    Car Audi=new Car();

  }


}
```

A Class is a 3-Compartment Box Encapsulating Data and Operations

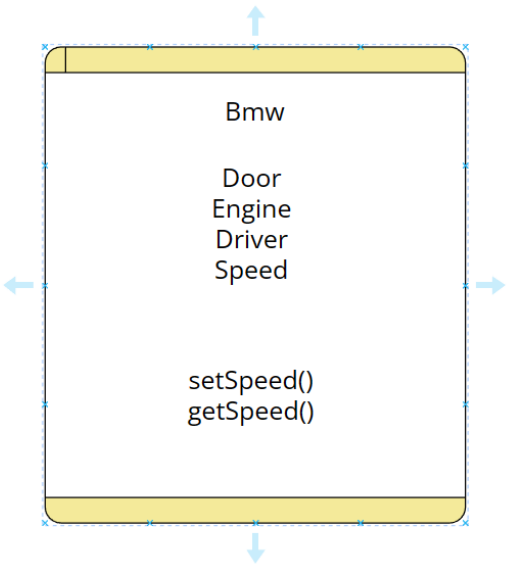


A class can be visualized as a three-compartment box, as illustrated:

1. *Name* (or identity): identifies the class.
2. *Variables* (or attribute, state, field): contains the *static attributes* of the class.

3. *Methods* (or behaviors, function, operation): contains the *dynamic behaviors* of the class.

In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.



1. A *class* is a programmer-defined, abstract, self-contained, reusable software entity that mimics a real-world thing.
2. A class is a 3-compartment box containing the name, variables and the methods.
3. A class encapsulates the data structures (in variables) and algorithms (in methods). The values of the variables constitute its *state*. The methods constitute its *behaviors*.
4. An *instance* is an instantiation (or realization) of a particular item of a class.

Class Naming Convention: A class name shall be a noun or a noun phrase made up of several words. All the words shall be initial-capitalized (camel-case). Use a *singular* noun for class name. Choose a meaningful and self-descriptive classname. For examples, `SoccerPlayer`, `HttpProxyServer`, `FileInputStream`, `PrintStream` and `SocketFactory`.

Creating Instances of a Class

To create *an instance of a class*, you have to:

1. **Declare** an instance identifier (instance name) of a particular class.
2. **Construct** the instance (i.e., allocate storage for the instance and initialize the instance) using the "`new`" operator.

```
public static void main(String[] args) {
    Car Bmw=new Car();
    Car Mercedes=new Car();
    Car Volvo=new Car();
    Car Audi=new Car();

  }
```

Dot (.) Operator

The *variables* and *methods* belonging to a class are formally called *member variables* and *member methods*. To reference a member variable or method, you must:

1. First identify the instance you are interested in, and then,
1. Use the *dot operator* (`.`) to reference the desired member variable or method.

```
public class Car {
  private String doors;
  private String engine;
  private String driver;
  private int speed;

  public void setSpeed(int speed){
    this.speed = speed;
  }
  public int getSpeed(){
    return speed;
  }

  public static void main(String[] args) {
    Car Bmw=new Car();
    Car Mercedes=new Car();

    Bmw.setSpeed(120);
    Mercedes.setSpeed(150);

  System.out.println(Bmw.getSpeed());
  System.out.println(Mercedes.getSpeed());
  }
}
```

Member Variables

A *member variable* has a *name* (or *identifier*) and a *type*; and holds a *value* of that particular type (as descried in the earlier chapter).

Variable Naming Convention: A variable name shall be a noun or a noun phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case), e.g., `fontSize`, `roomNumber`, `xMax`, `yMin` and `xTopLeft`.

```
[AccessControlModifier] type variableName [=initialValue];
[AccessControlModifier] type variableName-1 [=initialValue-1] [,type variableName-2 [=ini
tialValue-2]] ...;
```

```
private String doors,engine,driver;
private int speed;
```

Member Methods

A method (as described in the earlier chapter):

1. receives arguments from the caller,
2. performs the operations defined in the method body, and
3. returns a piece of result (or `void`) to the caller.

Method Naming Convention: A method name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). For example, `getArea()`, `setRadius()`, `getParameterValues()`, `hasNext()`.

```
[AccessControlModifier]returnType methodName([parameterList]) {
// method body or implementation
   ......
}
```

```
public void setSpeed(int speed) {
    this.speed = speed;
  }

  public int getSpeed() {
    return speed;
  }
```

Variable name vs. Method name vs. Class name

A **variable** name is a **noun**, denoting an **attribute**;

A **method** name is a **verb**, denoting an **action**.

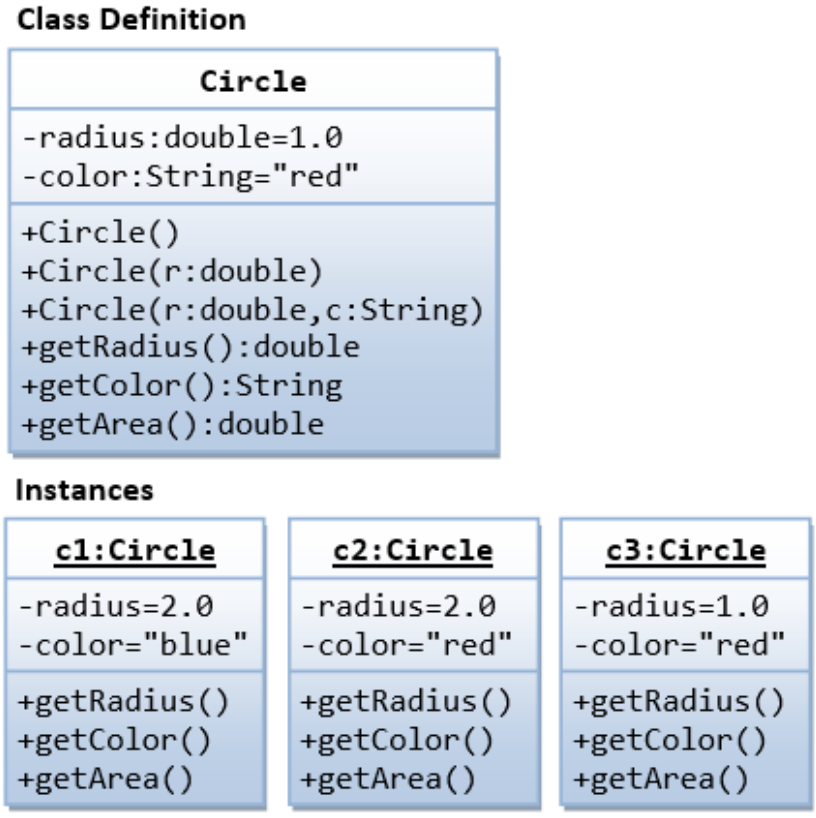
They have the **same naming convention** (the first word in lowercase and the rest are initial-capitalized). Nevertheless, we can easily distinguish them from the context.

Methods take **arguments** in **parentheses** (possibly zero arguments with empty parentheses),

Variables do not.

In this writing, methods are denoted with a pair of parentheses, e.g., `println()`, `getArea()` for clarity.

On the other hand, **class** name is a **noun beginning** with **uppercase**.



A class called `Circle` is defined as shown in the class diagram. It contains two `private` member variables: `radius` (of type `double`) and `color` (of type `String`); and three `public` member methods: `getRadius()`, `getColor()`, and `getArea()`.

Three instances of `Circle`s, called `c1`, `c2`, and `c3`, shall be constructed with their respective data members, as shown in the instance diagrams.

Constructors

A *constructor* is a special method that has the *same method name as the class name*. That is, the constructor of the class `Circle` is called `Circle()`. In the above `Circle` class, we define three overloaded versions of constructor `Circle(...)`. A constructor is used to *construct* and *initialize* all the member variables. To construct a new instance of a class, you need to use a special "`new`" operator followed by a call to one of the constructors.

A constructor method is different from an ordinary method in the following aspects:

- The *name* of the constructor method must be the same as the classname. By classname's convention, it begins with an uppercase (instead of lowercase for ordinary methods).
- Constructor has *no return type* in its method heading. It implicitly returns `void`. No `return` statement is allowed inside the constructor's body.
- Constructor can only be invoked via the "`new`" operator. It can only be used *once* to initialize the instance constructed. Once an instance is constructed, you cannot call the constructor anymore.
- Constructors are not inherited (to be explained later). Every class shall define its own constructors.

Default Constructor:

A constructor with no parameter is called the *default constructor*. It initializes the member variables to their default values. For example, the `Circle()` in the above example initialize member variables `radius` and `color` to their default values.

If we don’t implement any **constructor** in our class, the Java compiler inserts default constructor into our code on our behalf.

no-arg constructor:

Constructor with no arguments is known as **no-arg constructor**. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.

Parameterized constructor

Constructor with arguments(or you can say parameters) is known as **Parameterized constructor**.

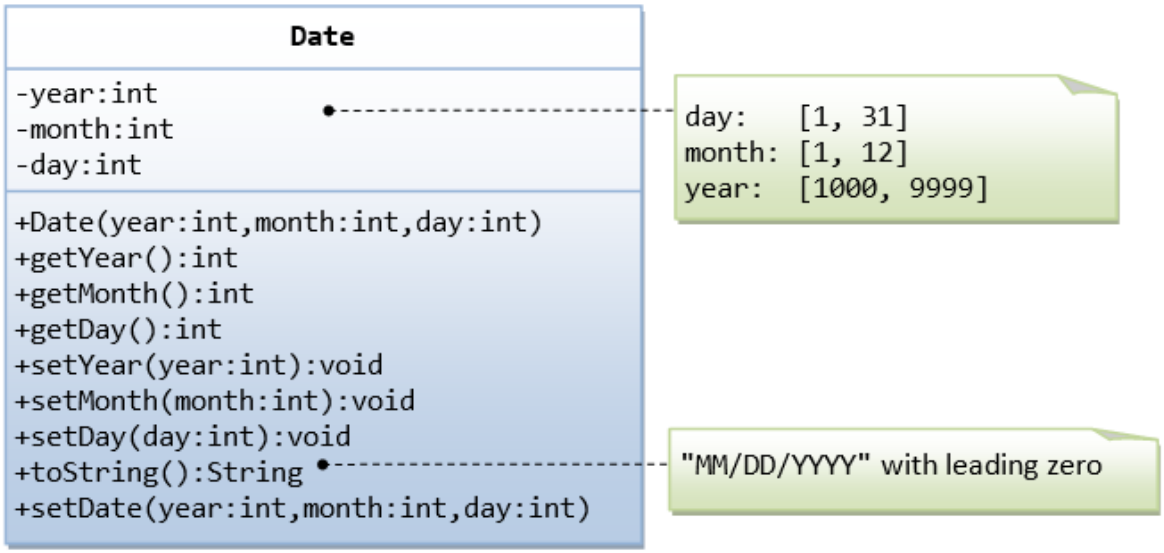
Constructor Chaining

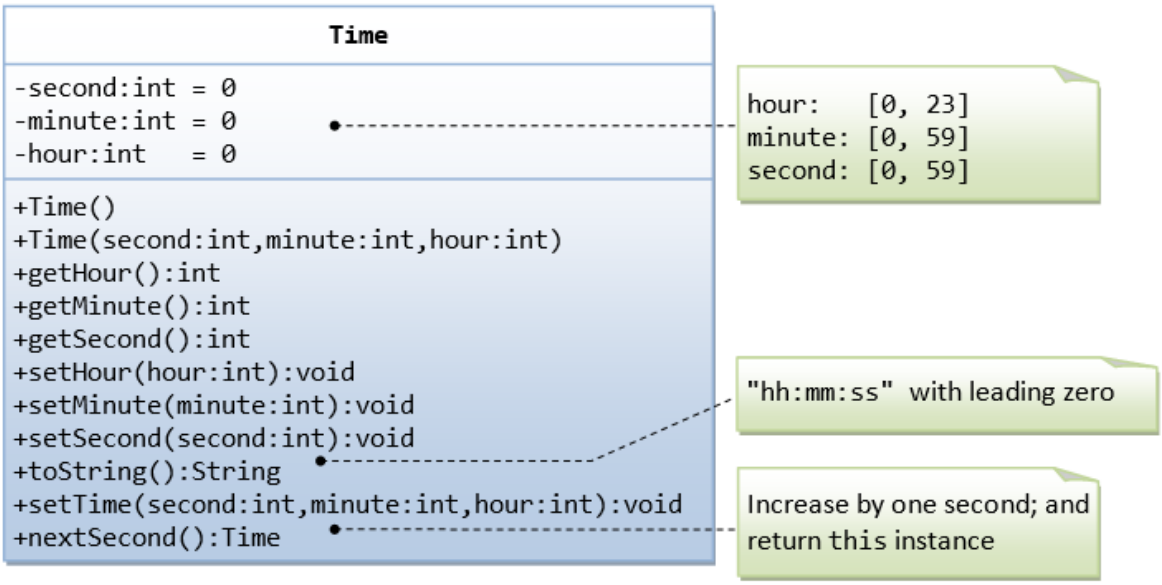
Calling a **constructor** from the **another constructor** of **same class** is known as **Constructor chaining**.

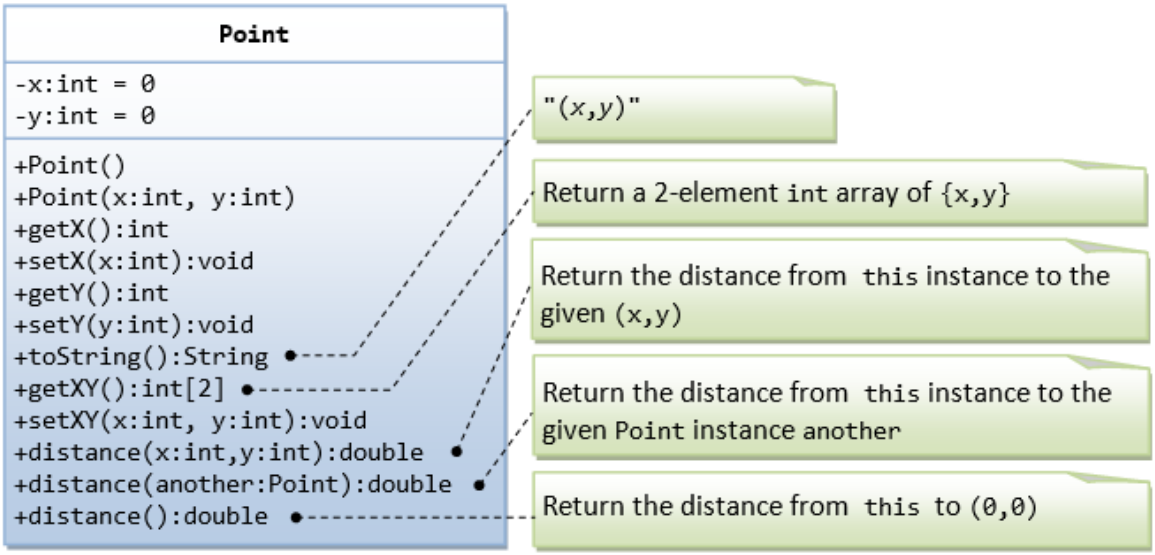
The real purpose of **Constructor Chaining** is that we can pass parameters through a bunch of different constructors,but only have the initialization done in a single place.

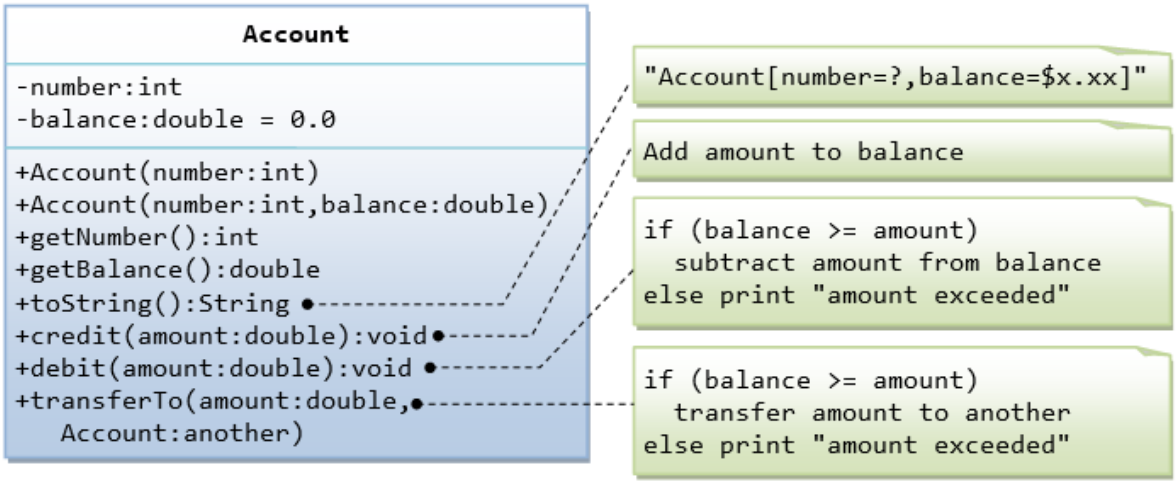
This allows us to maintain initializations from a single location, while providing multiple constructors to the user. If we don't chain, and two different constructors require a specific parameter, we will have to initialize that parameter twice, and when the initialization changes, we have to change it in every constructor, instead of just the one.

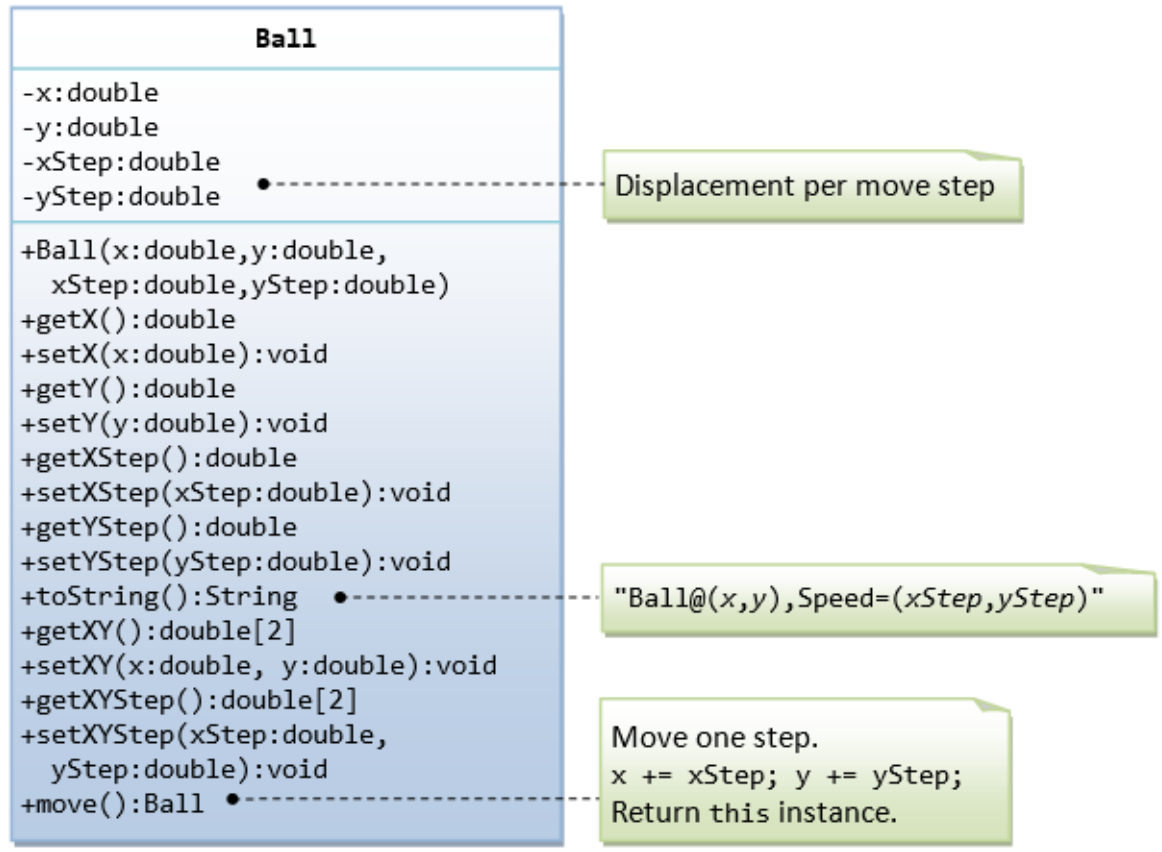
- **Within same class**: It can be done using **this()** keyword for constructors in the same class
- **From base class:** by using **super()** keyword to call the constructor from the base class.

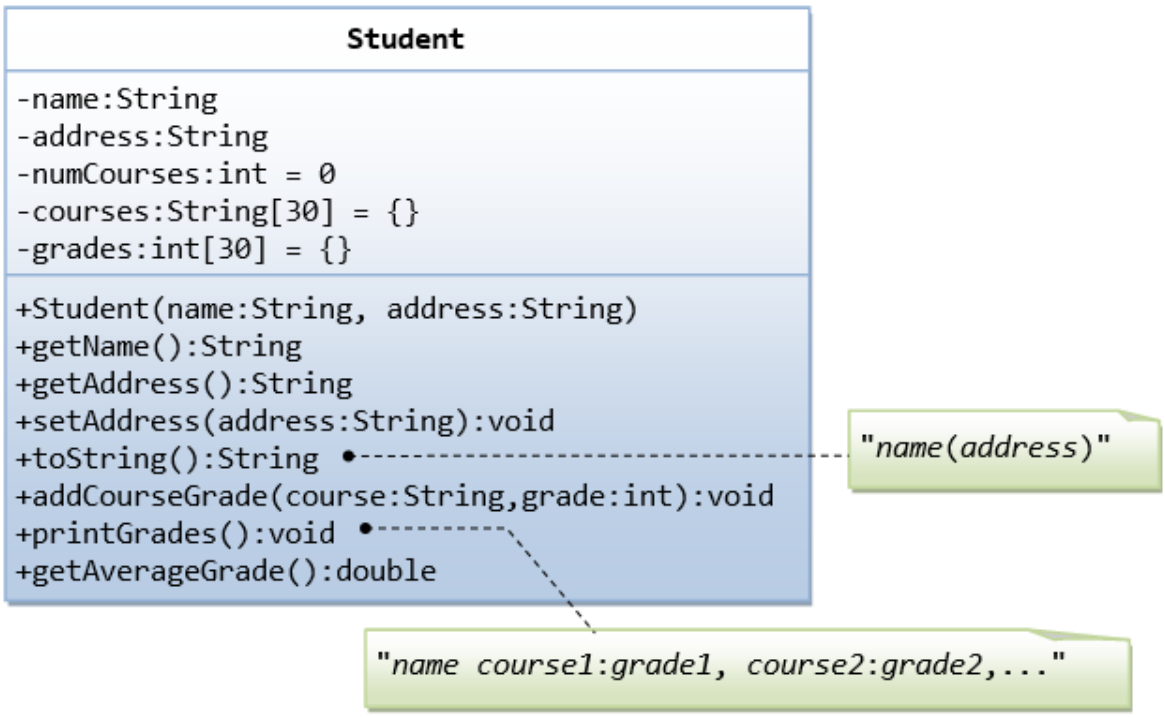



Time
-second:int = 0
-minute:int = 0
-hour:int = 0
+Time()
+Time(second:int,minute:int,hour:int)
+getHour():int
+getMinute():int
+getSecond():int
+setHour(hour:int):void
+setMinute(minute:int):void
+setSecond(second:int):void
+toString():String
+setTime(second:int,minute:int,hour:int):void
+nextSecond():Time
hour: [0, 23]
minute: [0, 59]
second: [0, 59]
"hh:mm:ss" with leading zero
Increase by one second; and return this instance


Point
-x:int = 0
-y:int = 0
+Point()
+Point(x:int, y:int)
+getX():int
+setX(x:int):void
+getY():int
+setY(y:int):void
+toString():String
+getXY():int[2]
+setXY(x:int, y:int):void
+distance(x:int,y:int):double
+distance(another:Point):double
+distance():double
"(x,y)"
Return a 2-element int array of {x,y}
Return the distance from this instance to the given (x,y)
Return the distance from this instance to the given Point instance another
Return the distance from this to (0,0)


Account
-number:int
-balance:double = 0.0
+Account(number:int)
+Account(number:int,balance:double)
+getNumber():int
+getBalance():double
+toString():String
+credit(amount:double):void
+debit(amount:double):void
+transferTo(amount:double,
Account:another)
"Account[number=?,balance=$x.xx]"
Add amount to balance
if (balance >= amount)
subtract amount from balance
else print "amount exceeded"
if (balance >= amount)
transfer amount to another
else print "amount exceeded"





+91 79808 81197