COMP20010

Data Structures and Algorithms I

Tutorial 2

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Data Types

Data types. A *data type* is a set of values and a set of operations on those values.

Abstract data types. An *abstract data type* is a data type whose internal representation is hidden from the client.

Objects. An *object* is an entity that can take on a data-type value. Objects are characterised by three essential properties: The *state* of an object is a value from its data type; the *identity* of an object distinguishes one object from another; the *behaviour* of an object is the effect of data-type operations. In Java, a *reference* is a mechanism for accessing an object.

Applications programming interface (API). To specify the behaviour of an abstract data type, we use an *application programming interface* (API), which is a list of *constructors* and *instance methods* (operations), with an informal description of the effect of each, as in this API for `Counter`:

Client. A client is a program that uses a data type.

Implementation. An implementation is the code that implements the data type specified in an API.

Data Types

Creating objects. Each data-type value is stored in an object. To create (or *instantiate*) an individual object, we invoke a *constructor* by using the keyword `new`. Each time that a client uses `new`, the system allocates memory space for the object, initialises its value, and returns a reference to the object.

```
Point2D p = new Point2D(x0, y0);
```

declaration

constructor

Data Types

Invoking instance methods. The purpose of an instance method is to operate on data-type values. Instance methods have all of the properties of static methods: arguments are passed by value, method names can be overloaded, they may have a return value, and they may cause side effects. They have an additional property that characterizes them: *each invocation is associated with an object*.

```
double dist = p1.distanceTo(p2);
```

result

instance method call

Data Types

Objects as arguments. You can pass objects as arguments to methods. Java passes a *copy* of the argument value from the calling program to the method. This is known as *pass by value*. If you pass a reference to an object of type `Counter`, Java passes a *copy* of that reference. The method cannot change the original reference, but it can change the value of the object.

```
public double distanceTo(Point2D other) {
    other.moveTo(0.0, 0.0);
    //
}
```

can change the object by calling member functions

```
double dist = p1.distanceTo(p2);
```

```
public double distanceTo(Point2D other) {
    other = new Point2D(0.0,0.0);
    //
}
```

not allowed reassign object

Implementing Abstract Data Types

- *Instance variables*. To define data-type values (the state of each object), we declare instance variables in much the same way as we declare local variables. There are numerous values corresponding to each instance variable (one for each object that is an instance of the data type). Each declaration is qualified by a *visibility modifier*. In ADT implementations, we use `private`, using a Java language mechanism to enforce the idea that the representation of an ADT is to be hidden from the client, and also `final`, if the value is not to be changed once it is initialised.

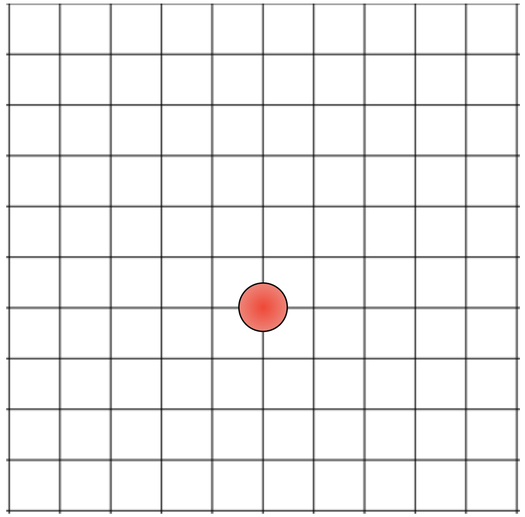
- *Constructors*. The constructor establishes an object's identity and initializes the instance variables. Constructors always share the same name as the class. We can overload the name and have multiple constructors with different signatures, just as with methods. If no other constructor is defined, a default no-argument constructor is implicit, has no arguments, and initialises instance values to default values. The default values of instance variables are 0 for primitive numeric types, `false` for `boolean`, and `null`.

- *Instance methods*. Instance methods specify the data-type operations. Each instance method has a return type, a *signature* (which specifies its name and the types and names of its parameter variables), and a *body* (which consists of a sequence of statements, including a *return* statement that provides a value of the return type back to the client). When a client invokes a method, the parameter values (if any) are initialised with client values, the statements are executed until a return value is computed, and the value is returned to the client. Instance methods may be *public* (specified in the API) or *private* (used to organise the computation and not available to clients).

Data Type: Point2D

We want to represent a point in the 2 dimensional place

Common in graphics applications

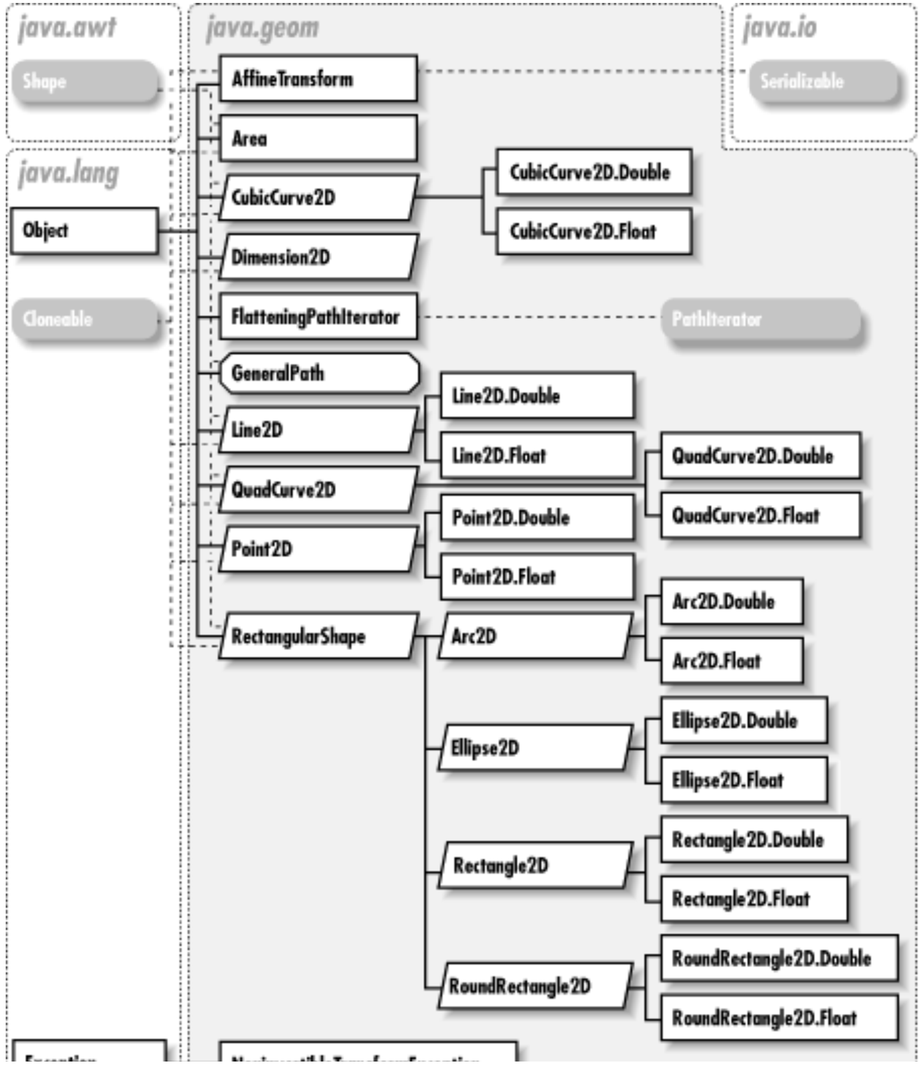


What variables should we use? (int, double, float, long?)

what methods should it have?

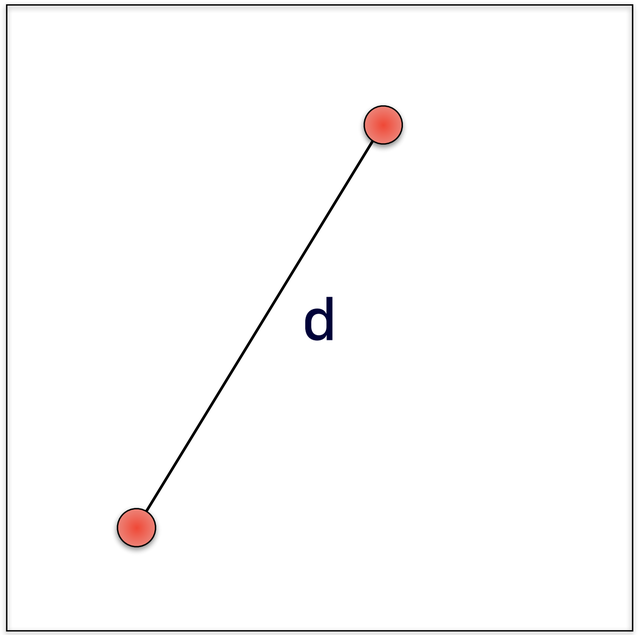
What constructors should it have?

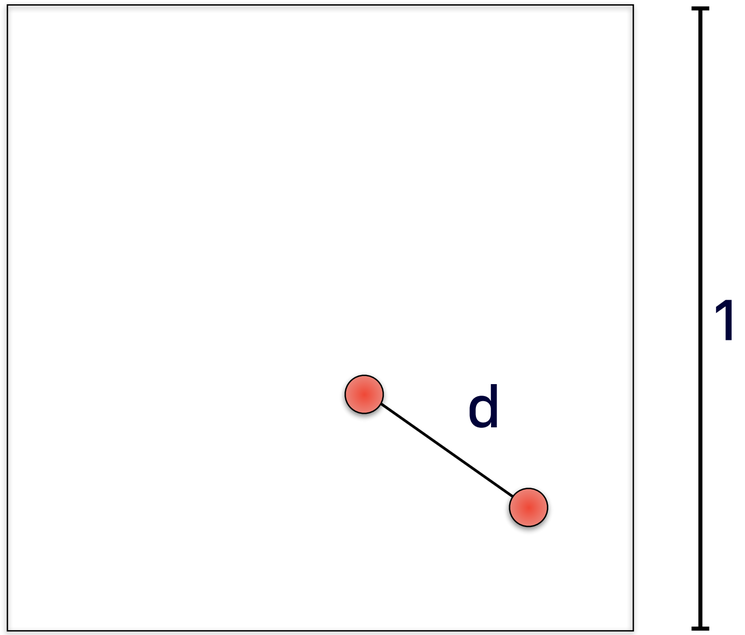
Depends what me want to use it for?



Problem Statement:

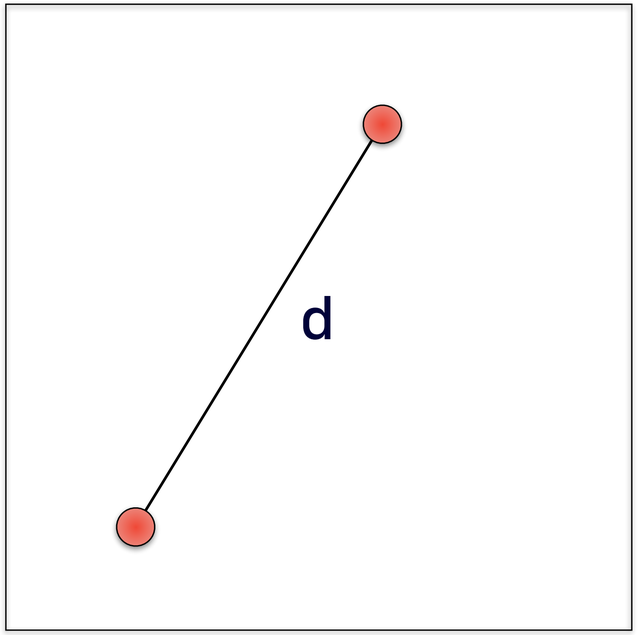
- Imagine placing two points at random in a square.
- What is the average distance between these two points?
- Guess?

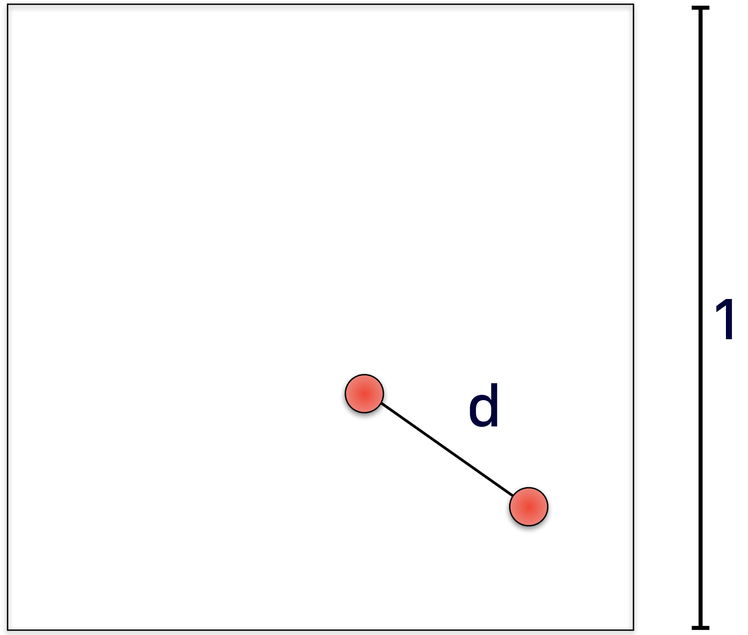




Task:

- Find the average distance by writing a Java program to generate points at random and compute the distance between them.
- Implementing the Point2D data type and the necessary methods (p1.distanceTo(p2)).





You can start with this code:

```
public class Point2D {

    public Point2D(double x, double y) {
   // TODO
    }

    public double x() {
      // TODO
    }

    public double y() {
   // TODO
    }


    public double distanceTo(Point2D that) {
   // TODO
    }

    public String toString() {
   // TODO
    }

    public static void main(String[] args) {
        int x0 = 1;
        int y0 = 2;
        Point2D p = new Point2D(x0, y0);
        System.out.println(p);
    }
}
```

Outline:

Create an array of Point2D objects. *How many?*

Initialise the array of objects to random values.

For each pair of points compute the distance between them.

Compute the average of the distances.

Bonus: How does the average distance depend on the number of points we test? Report the average distance for different number of points.