Classes

Structures more complex than primitive types are defined with the help of classes and in object-oriented manner. The mission of explaining the concepts of object-oriented design is impossible in the format of a small section: the subject deserves a separate book, and there are a lot of them already written. All we can do here is give you a feeling of what object-orientedness is about, introduce such fundamental terms as class, method, object, instance, subclass, and inheritance, and show how Java supports object-oriented design. You should not try to learn or fully understand the code fragments in this section. It will be sufficient if, having read this section, you will know that, for example, a statechart in your model is an instance of AnyLogic class `Statechart` and you can find out its current state by calling its method: `statechart.getActiveSimpleState()`.

Class as grouping of data and methods. Objects as instances of class

Consider an example. Suppose you are working with local map and use coordinates of locations and calculate distances. Of course you can remember two double values `x` and `y` for each location and write a function `distance( x1, y1, x2, y2 )` that would calculate distances between two pairs of coordinates. But it is a lot more elegant to introduce a new entity that would group together the coordinates and the method of calculating distance to another location. In object-oriented programming such entity is called a *class*. Java class definition for the location on a local map can look like this:

```
class Location {

    //constructor: creates a Location object with given coordinates
    Location( double xcoord, double ycoord ) {
        x = xcoord;
        y = ycoord;
    }

    //two fields of type double
    double x; //x coordinate of the location
    double y; //y coordinate of the location

    //method (function): calculates distance from this location to another one
    double distanceTo( Location other ) {
        double dx = other.x - x;
        double dy = other.y - y;
        return sqrt( dx*dx + dy*dy );
    }

}
```

As you can see, a class combines data and methods that work with the data.

Having defined such class, we can write very simple and readable code when working with the map, like this:

```
Location origin = new Location( 0, 0 ); //create first location

Location destination = new Location( 250, 470 ); //create second location

double distance = origin.distanceTo( destination ); //calculate distance
```

The locations `origin` and `destination` are *objects* and are *instances* of the class `Location`. The expression `new Location( 250, 470 )` is a *constructor call*, it creates and returns a new instance of the class `Location` with the given coordinates. The expression `origin.distanceTo( destination )` is a *method call* – it asks the object `origin` to calculate the distance to another object `destination`.

If you declare a variable of a non-primitive type (of a class) and do not initialize it, its value will be set to `null` (`null` is a special Java literal that denotes "*nothing*"). Sometimes you explicitly assign `null` to a variable to "forget" the object it referred to and to indicate that the object is missing or unavailable.

```
Location target; //a variable is declared without initialization. target equals null
target = warehouse; //assign the object (pointed to by the variable) warehouse to target
//now target and warehouse point to the same object
…
target = null; //target forgets about the warehouse and equals null again
```

Inheritance. Subclass and super class

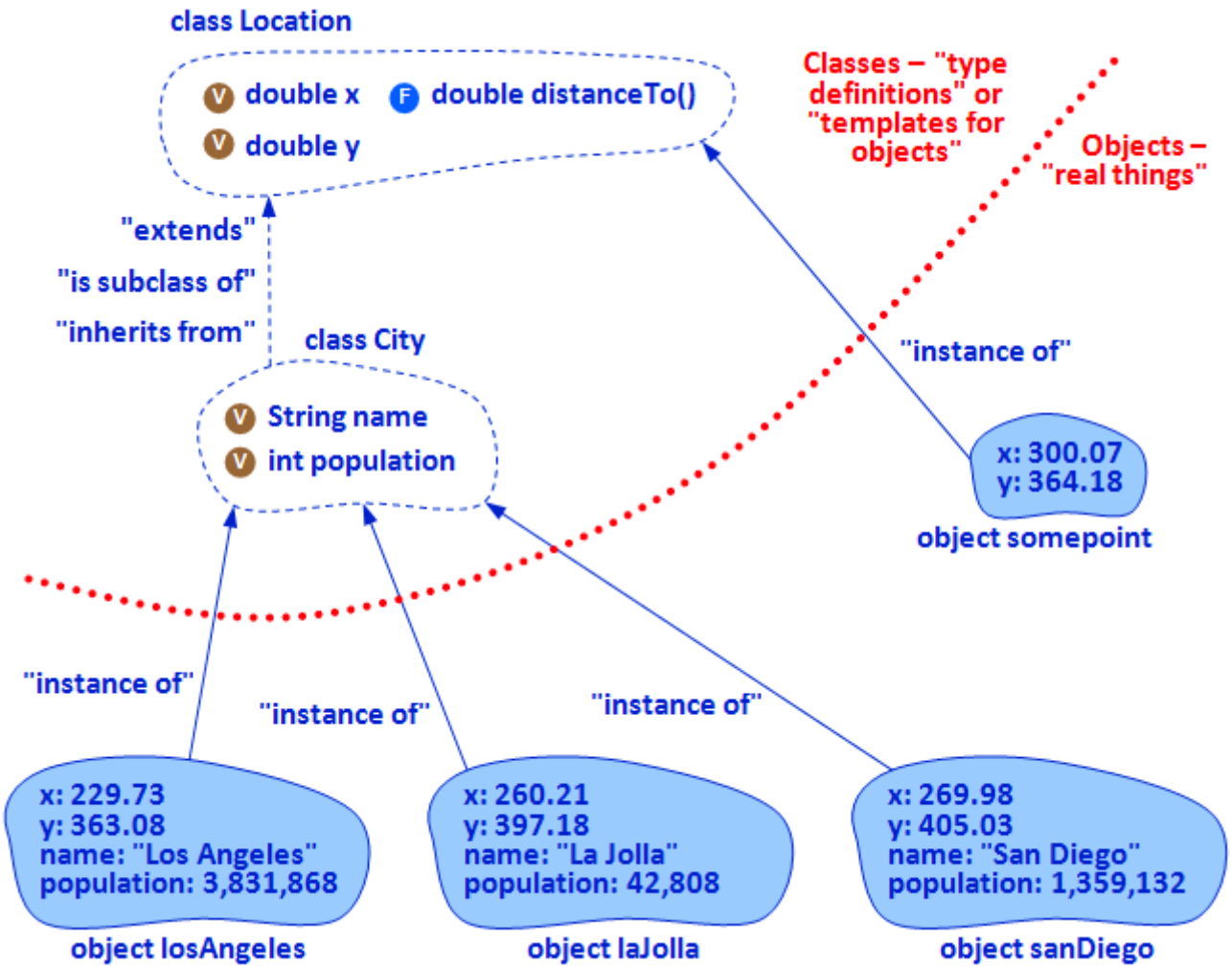
Now suppose some locations in your 2D space correspond to cities. A city has a name and population size. To efficiently manipulate cities we can extend the class `Location` by adding two more fields: `name` and `population`. We will call the new agent `City`. In Java this will look like:

```
class City extends Location { //declaration of the class CIty that extends the class Location

    //constructor of class City
    City( String n, double x, double y ) {
        super( x, y ); //call of constructor of the super class with parameters x and y
        name = n;
        //the population field is not initialized in the constructor - to be set later
    }

    //fields of class City
    String name;
    int population;

}
```



Classes and inheritance

`City` is called a *subclass* of `Location`, and `Location` is correspondingly a *super class* (or *base class*) of `City`. `City` *inherits* all properties of `Location` and adds new ones. See how elegant is the code that finds the biggest city in the range of 100 kilometers from a given `point` of class `Location` (we assume that there is a collection `cities` where all cities are included):

```
int pop = 0; //here we will remember the largest population found so far
City biggestcity = null; //here we will store the best city found so far

for( City city : cities ) { //for each city in the collection cities
    if( point.distanceTo( city ) < 100 && city.population > pop ) { //if best so far
        biggestcity = city; //remember it
        pop = city.population; //and remember its population size
    }
```

```
}
traceln( "The biggest city within 100km is " + city.name ); //print the search result
```

Notice that although `city` is an object of class `City`, which is "bigger" than `Location`, it still can be treated as Location when needed, in particular when calling the method `distanceTo()` of class `Location`.

This is a general rule: an object of a subclass can always be considered as an object of its base class.

How about vice versa? You can declare a variable of class `Location` and assign it an object of class `City` (a subclass of `Location`):

```
Location place = laJolla;
```

This will work. However, if you will then try to access the population of place, Java will signal an error:

```
int p = place.population; //error: "population cannot be resolved or is not a field"
```

This happens because Java does not know that `place` is in fact an object of class `City`. To handle such situations you can:

- test whether an object of a certain class is actually an object of its particular subclass by using the operator `instanceof`: *<object>* `instanceof` *<class>*
- "*cast*" the object from a super class to a subclass by writing the name of the subclass in parenthesis before the object: (*<class>*)*<object>*

A typical code pattern is:

```
if( place instanceof City ) {
    City city = (City)place;
    int p = city.population;
    …
}
```

Classes and objects in AnyLogic models

Virtually all objects that you use to create your models are instances of AnyLogic Java classes. Please refer to AnyLogic class reference to find the Java class names for most model elements you work with. Whenever you will need to find out what can you do with a particular object using Java, you should find the corresponding Java class in *AnyLogic API Reference* and look through its methods and fields.