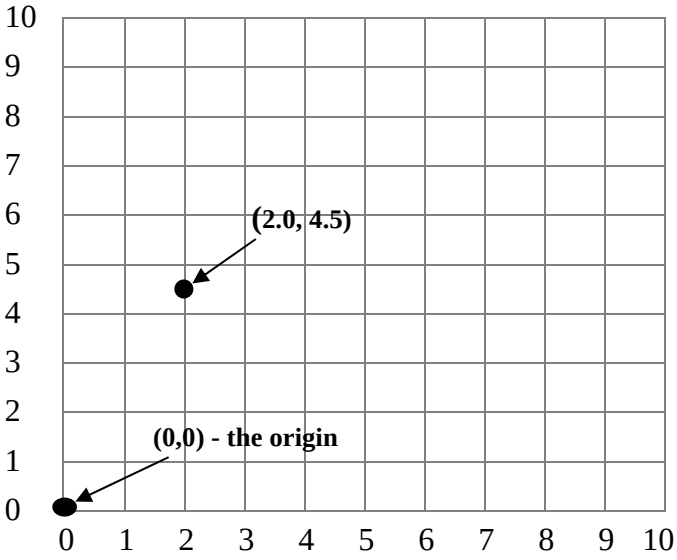
3 Classes and Objects

1 Classes as compound types

If we want to represent a sum of money in a program we can use an integer. If we want to represent a person's name, we can use a string. But if we want to represent a point on a grid, we need a pair of numbers:



Java does not supply a type corresponding to pairs of items, but it gives us a mechanism for making such types. A *class* is in essence a mechanism for introducing a new type composed from simpler types. For example, a type to represent points will have two components, each one a number. The upper point in the diagram above has components 2.0 – the x-component, and 4.5 – the y-component. Java supplies us with so-called "elementary" types, namely the integers, booleans, characters, and reals. All other types have to be introduced by the programmer as needed, and this is done using classes.

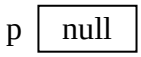
To introduce a type for representing points, we write a class such as

```
class Point {
    double x, y;
}
```

This text defines a new type called `Point`. Note that it introduces a *type* – it does not introduce any variables, notwithstanding the presence of declaration statements for x and y. It should be viewed as a template from which points will be created, as we shall see. The names `x` and `y` are chosen arbitrarily – we could equally well have called them `u` and `u`, say. The class definition as written above is placed either before or after the class containing main(). The name Point has the same status as int or String – it is the name of a type. The statement

```
Point p;
```

introduces p as a variable of type Point. (For reasons that will become clear later, it is more accurate to say that p is of type "reference to Point", but that is a minor subtlety.) "Point p;" is a declaration statement just like "int n;", and can be written wherever "int n;" might be written. Initially variable p is given the special value *null* by default; it may be pictured as:



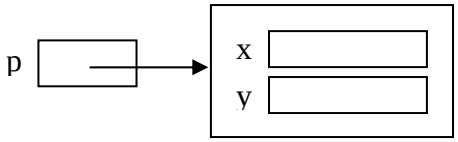
In general, the members of a class type are called "objects". More particularly, the members of class Point are called "Point objects", or "instances" of Point. To manipulate points in a program we create Point objects, one for each point we want to handle. Every object is made of components as specified in the class definition. For example, objects of type Point will have two components, each of them being a variable of type double. The constituent variables of a class are called its "instance variables"; for example, x and y are the instance variables of class Point. The expression "new Point()" creates an object of type Point. It usually occurs in an assignment statement:

```
p = new Point();
```

This does **two** things (remember that – **two** things):

(i) It creates a new Point object.
(ii) It places a reference to it in p.

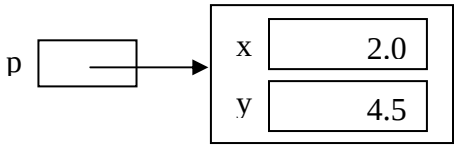
The effect can be visualised thus:



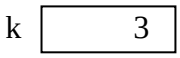
The component parts of the point just created are referred to as p.x and p.y, respectively. Every time you create an instance of a class, you create new versions of the instance variables. For each object created, its instance variables can be treated as regular variables that can be assigned values. For example, the effect of the assignments

```
p.x = 2.0;  p.y = 4.5;
```

can be pictured as



We say that variable p is of type *reference to* Point because, as the picture indicates, variable p does not contain the point (2.0, 4.5) but a *reference* to it. In contrast, when an integer variable k, say, is assigned the value 3, its state may be depicted as follows:



-- observe that variable k literally contains 3.

The following trivial program illustrates the use of class Point. It calculates the distance between a point (chosen arbitrarily as (2.0, 4.5)) and the origin (i.e. the point (0,0)). The program uses a standard mathematical formula for distance from the origin; if you are not familiar with it just take it on trust (the function Math.sqrt(x) yields the square root of x).

```
class Point {
    double x, y;
}

class PointDemo {
```

```
    public static void main(String args[] ) {
        Point p;                          // declare p
        p = new Point();                  // create a Point object, accessed via p
        p.x = 2.0;  p.y = 4.5;            // fill in the component values
        double dist = Math.sqrt(p.x*p.x+ p.y*p.y); // distance of p from origin
        System.out.println("The point is " + dist + " from the origin.");
    }
}
```

Remember that the names we choose for the name of a class and its constituent components are arbitrary; we only need to stick throughout the program with whatever names we've chosen. For example, the above program might well have been written as

```
class PointClass {
    double u, v;
}

class PointDemo {

    public static void main(String args[] ) {
        PointClass p;                     // declare p
        p = new PointClass ();            // create a Point object, accessed via p
        p.u = 2.0;  p.v = 4.5;            // fill in the component values
        double dist = Math.sqrt(p.u*p.u+ p.v*p.v); // distance of p from origin
        System.out.println("The point is " + dist + " from the origin.");
    }
}
```

Classes are fundamental in Java, and it is important that you get off to a good start. Make sure you understand what is meant by:

(i) The class definition (the first three lines of the program above);
(ii) Declaring a variable for holding (a reference to) an instance of the class (e.g. `Point p`);
(iii) Creating an instance of the class (e.g. `new Point()`);
(iv) Placing a reference to it in a variable (e.g. the assignment in `p = new Point()`);
(v) Initialising the instance variables (e.g. `p.x = 2.0; p.y = 4.5;`).

The above program is trivial, and can be easily re-written to avoid the use of classes. Classes are not terribly useful until we combine them with methods, which we shall do shortly. For the moment, we are concentrating on the technical rudiments. Here are some further examples of defining classes. A class for dates:

```
class Date {
    int day, month, year;
}
```

A class for students:

```
class Student {
    String name;
    String[] courses;
    boolean isMale;
    int age;
}
```

A class for CDs:

```
class CD {
    String artist;
    String title;
    String[] trackTitles;
    int yearOfRelease;
}
```

It is convention among Java programmers to start class names with an upper case letter, even though this is not required by the language rules.

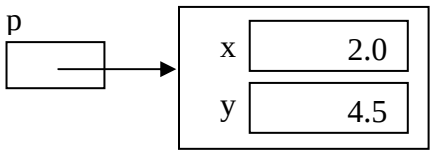
2 Using classes with static methods

When we use a class to introduce a new type, we can use the class type in any context where we can use the given types such as `int` or `bool`. In particular we can use them with methods to indicate a return type in a function, or to indicate the type of a parameter.

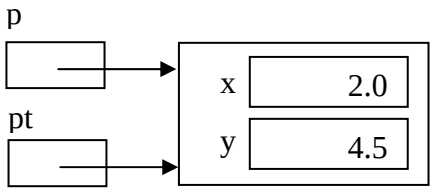
For example, here is an another version of the previous program to compute the distance of a point from the origin. It illustrates that the type of parameters may be class names:

```
class Point {
        double x, y;
}

class PointDemo {

        static double distance(Point pt) { // distance from the origin
                return(Math.sqrt(pt.x*pt.x+ pt.y*pt.y));
        }

        public static void main(String args[] ) {
                Point p = new Point();
                p.x = Console.readDouble();
                p.y = Console.readDouble();
                System.out.println("Distance from origin: " + distance(p));
        }
}
```

Parameters of class types work exactly as parameters of basic types. In this case the main program creates and initialises p with a point object, say:



The system begins the invocation of distance (p) by initialising the parameter pt with the value of the associated argument, i.e. p, after which we have

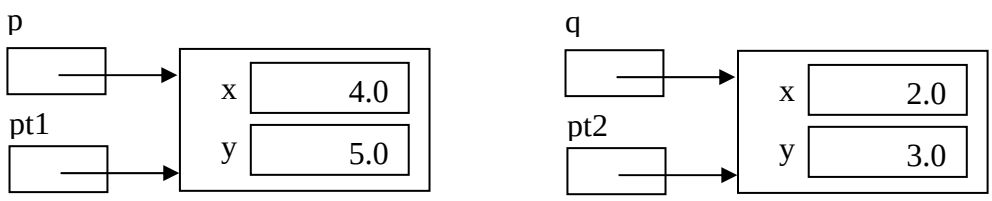


The body of distance() now works on the object referenced by pt, which is of course the object that the main program created and initialised.

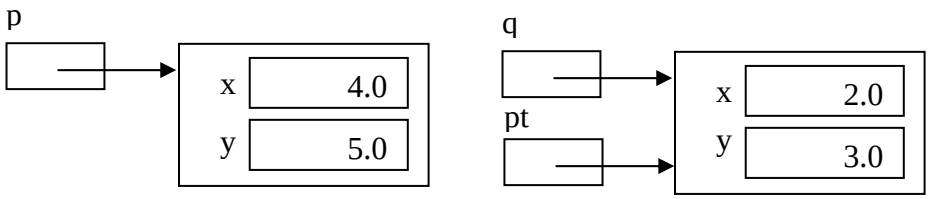
Here is a slightly different program, one that determines which of two points lies closer to the origin. It illustrates that the return type of a method may be a class type.

```
class Point {
        double x, y;
}

class PointDemo01 {

        static Point closerOf(Point pt1, Point pt2) {
                double d1 = Math.sqrt(pt1.x*pt1.x+ pt1.y*pt1.y);
                double d2 = Math.sqrt(pt2.x*pt2.x+ pt2.y*pt2.y);
                if (d1<d2) return pt1;
                else  return pt2;
        }

        public static void main(String args[]) {
                Point p = new  Point();
                p.x = Console.readDouble();  p.y = Console.readDouble();
                Point q = new Point();
                q.x = Console.readDouble();  q.y = Console.readDouble();
                Point pt = closerOf(p,q);
                System.out.print(pt.x + " " + pt.y);
        }
}
```

Let's take a closer look at the operation of statement Point pt = closerOf(p,q);. When after the main program has invoked closerOf(), the situation is, say:



closerOf will terminate in this case by effecting a return of pt2. This does NOT return the object referenced by pt2, but returns the reference itself. So the situation in main() after return from closerOf() is:



Note that we don't write `Point pt = new Point(); pt = closerOf(p,q);` but write `Point pt = closerOf(p,q);` -- there is no need to create a point object in this case.

Actually, it is much more common that we use dynamic (or instance) methods when we manipulate objects, rather than static methods as above. We will study instance methods shortly.

3 The big traps

The no-object trap

A declaration of a variable of a class type does not create an instance of the class. For example, the declaration

```
Point p;
```

only introduces a variable of type reference to Point, but it does not create a Point object. Hence the code

```
Point p; p.x = 1.2;
```

is erroneous because p.x does not exist at that point (p has the initial default value of *null*). Remember that variable p does not refer to an object until it is assigned a reference to an object. This is typically done by executing `p = new Point()` which leaves p referring to a newly created object. However, variables of a class type can be made refer to an object in other ways, as in the statement Point pt = closerOf(p,q); in the preceding program.

The needless-object trap

Sometimes, we needlessly create an object when a variable of a class type is declared, as illustrated by the following example. Suppose we have two point objects referenced by variables `p` and `q`:

```
Point p = new Point(); p.x = 1.2;  p.y = ...;
```

```
Point q = new Point(); q.x = ...;  q.y = ...;
```

Suppose that later in the program, we want to interchange `p` and `q`; for this we need a temporary variable of type `Point`. It is tempting to write:

```
Point t = new Point(); // silly!
t = p; p = q; q = t;
```

Although this code will work, it is silly to initialise `t` by creating a new Point object and placing a reference to it in `t` – because in the very next statement we assign `p` to `t`, and so the object we created plays no role at all. Instead you should introduce `t` using either

```
Point t;
```

or

```
Point t = null;
```

The instance variable trap

Whenever you refer to an instance variable, you must indicate the particular object in which the variable sits. The following code contains errors often made by beginners:

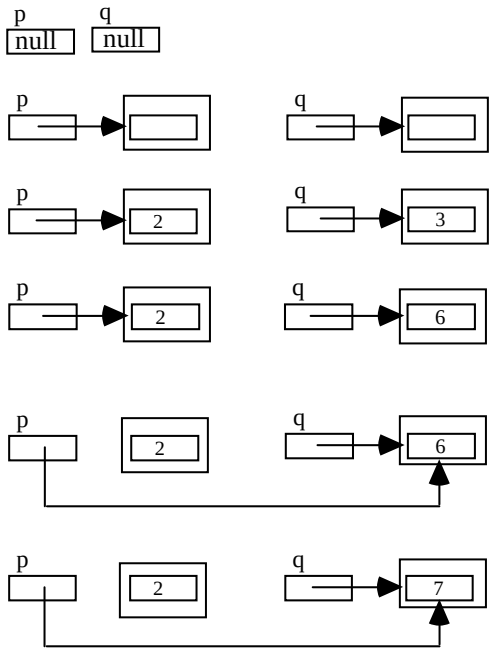
```
class MyClass {
        int x;
        ...;
}

class MainClass {
    public static void main(String[] args) {
        MyClass p = new MyClass (); // ok
        ....
        p.x = 7; // ok
        MyClass.x = 5; // Error!
        x = 3; // Error!
        .....
    }
}
```

It is crucial to get it absolutely clear that instance variables belong to objects, and do not exist merely by their occurrence in the class definition. Let x be an instance variable declared in class MyClass: *No variables called x are created until the programmer creates an object of type MyClass, and as many variables x are created as objects of type MyClass.* Whenever we refer to variable x, we must indicate *which* x, and we do this by prefixing it with a reference to the object of which it is a component (as in `p.x` where `p` is a variable of type `MyClass`).

4 Orphans and aliases

An orphan is an object which is no longer accessible to the program because we no longer retain a reference to it. Orphans arise naturally. The memory they occupy is released for later use by a component of the Java runtime system called the *garbage collector* which runs in the background. We illustrate with a (completely useless) program:

```
class Triv {
      int x;
}

class Silly {

  public static void main(String[] s) {
   Triv p, q;


   p = new Triv(); q = new Triv();


   p.x = 2; q.x = 3;


   q.x = p.x + 4;


   p = q;


   q.x = 7;


   System.out.print(p.x);

}
```



7 appears on the screen

The assignment p = q above creates an "orphan" (the object containing 2); the memory it occupies will be released eventually by the garbage collector.

In contrast to orphans (for which we have no access), some objects may have many access paths. When there is more than one way to access an object, the object is said to be *aliased*. In the trivial program above, for example, the object containing 7 can be accessed via variables p or q and so is aliased. Usually aliasing arises intentionally and presents no problems. However, occasionally it catches us out as explained below.

The aliasing trap

Recall that integer variables actually *contain* their integer values, and similarly for reals, chars, booleans, and floats. All other types of variable contain *references to* objects (references are also called "pointers" or "handles"). It is important to understand the distinction between them, because it affects the way assignment behaves. Consider the following code fragment

```
int j, k
j = 3; k = 5;
j = k; // integer copied!
k++;
System.out.println(j);
```

Obviously this code displays 5 – the assignment k++ has no effect on the value of j. Now consider the following which is similar to the above, but involves assignment of objects rather than integers

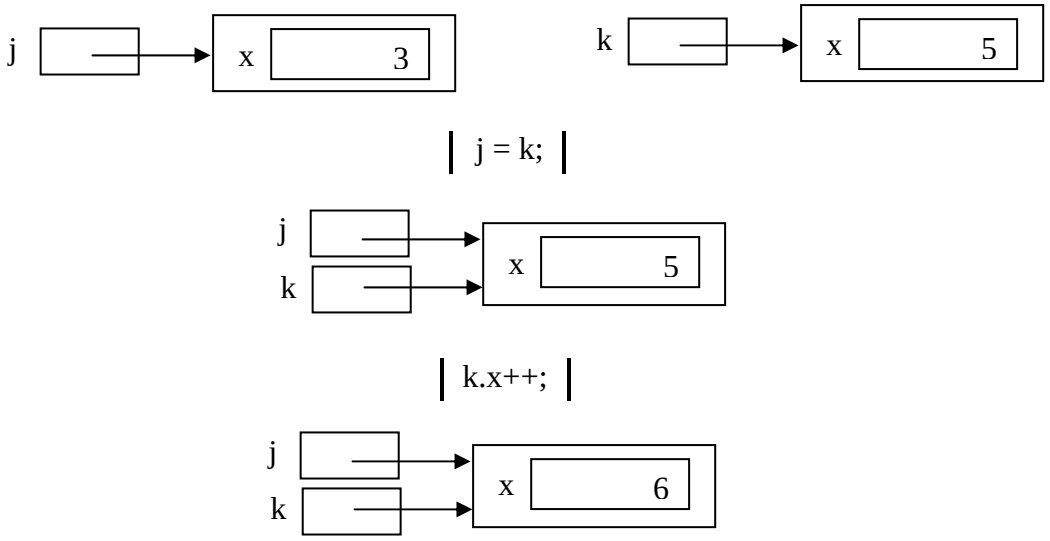
```
class Triv {
    int x;
}
....
Triv j, k;
j = new Triv(); j.x = 3;
k = new Triv(); k.x = 5;
j = k; // reference copied!
k.x++;
System.out.print(j.x);
```

Surprisingly for some, 6 is output rather than 3. The point to understand is that the assignment `j=k;` is effected by copying references, not values, leaving j and k referring to the same object –

see the picture below. Subsequently, any change to the state of the object accessed via j is concomitantly a change to the object accessed via k. If you do not want this, then the assignment j = k should be replaced with code which makes an exact copy of the object referenced by k, and assigns a reference to the new object to j:

```
j = new Triv(); j.x = k.x;  // new copy of k's object created!
```

Actually, it turns out in practice that making copies of objects is not much needed – copying references usually suffices. We will see a more systematic way to copy objects later.

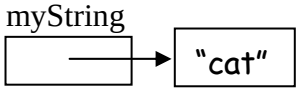


Strings are immutable objects

It so happens that strings are treated as objects in Java. The `String` type in Java is a built-in class, and hence variables of type `String` contain not an actual string but a reference to a string. For example, the declaration

```
String myString = "cat";
```

creates the situation depicted below:

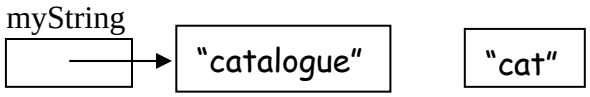


Hence, a variable of type `String` may be assigned the value *null*.

You never have to worry about inadvertently aliasing strings. This is because strings are "immutable", i.e. we can never change the value of a string. Of course, we can write an assignment such as

```
myString = myString + "alogue";
```

This does indeed change the string referred to by `myString`, but the change is effected not by modifying the original string, but by creating a new string similar to the original one but with "alogue" appended. The situation can be depicted as follows:



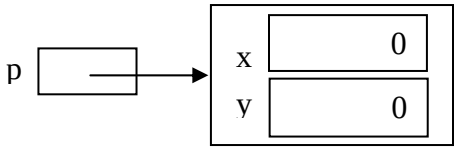
The original string object (containing "cat") becomes an orphan, which will eventually be discarded by the garbage collector.

5 Constructors

No-args constructors

For a class called `MyClass`, say, the phrase `new MyClass()` is an expression which yields a reference to a freshly created object of type `MyClass`. `MyClass()` is an example of a "constructor". Whenever we introduce a class, the system automatically provides a default constructor for it, as typified by `MyClass()`above.

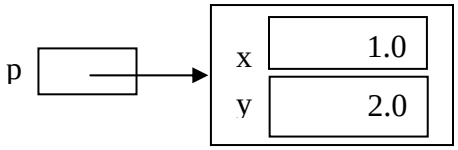
Instances variables are always given initial values by the constructor, according to certain default rules – e.g. integer components of objects are initialised to 0. For example, the effect of executing `Point p = new Point()` can be visualised as:



It is possible to change the default initial values, as follows:

```
class Point {
    double x = 1.0;
    double y = 2.0;
}
```

Now the effect of executing `Point p = new Point()` can be visualised as:



There is another way to provide specific initial values for instance variables: by writing your own constructor. (In contrast, the constructors supplied automatically by the system are said to be *default no-args constructors*). The following example illustrates a programmer-supplied constructor:

```
class Point {
    double x, y;

    Point() { // constructor with no parameters
            x = 1.0; y = 2.0;
    }

}
```

The effect of executing `Point p = new Point()` is exactly as in the preceding diagram.

Constructors with parameters

We write constructors just as we write methods, but *there is no return type in the header, nor any return statements in the body*. They *must* have the same name as the class. They may have parameters, just like methods, as in:

```
class Point {
    double x, y;

    Point(double xval, double yval) { // constructor with two parameters
            x = xval; y = yval;
    }

}
```

In this case, we must explicitly supply the desired initial values for the instance variables each time we construct the object, as in

```
p = new Point(2.5, 3.14);
```

The effect of this statement is to construct an object of type `Point`, initialise its `x` and `y` instance variables to 2.5 and 3.4, respectively, and assign a reference to the object to `p`. This is illustrated in the following trivial program:

```
class Point {
    double x, y;

    Point(double xval, double yval) { // constructor with two parameters
            x = xval; y = yval;
    }

}

class PointDemo {
    public static void main(String args[] ) {
        Point p = new Point(2.0,4.5);
        double dist = Math.sqrt(p.x*p.x+ p.y*p.y); // distance of p from origin
        System.out.println("The point is " + dist + " from the origin.");
    }
}
```

Remember: for constructors *there is no return type in the header, nor any return statements in the body*.

Class definitions can be composed using a combination of specifying default initial values for instance variables, and supplying zero or more constructors with varying numbers of arguments. This is illustrated below:

```
class Point {
    double x = 5.0; // default initial value of x components is 5.0
    double y;

    Point(double yval) {
            y = yval;
    }

    Point(double xval, double yval) {
            x = xval; y = yval;
```

```
    }
}

class ConstructorsTest {

    public static void main(String args[] ) {
            Point q, r;
            q = new Point(4.5); // q initialised to (5.0, 4.5)
            r = new Point(3.41, 4.5); // r initialised to (3.41, 4.5)
            .....
    }
}
```

The various constructors in a class definition must differ from one another in the number and types of their parameters. Constructors may invoke methods, whether they are defined in the same class or not. Actually, it is possible for a constructor to call another constructor within its class. The called constructor is invoked as though it were a void method (i.e. a procedure), and the invocation must be the *first* action in the calling constructor. This facility is not much used in small programs.

The default constructor trap

If you define your own constructor(s), then the default no-args constructor is no longer made available. If you really need it you have to write it, as for example in :

```
class Point {
    double x, y;

    Point(double xval, double yval) { // constructor with two parameters
            x = xval; y = yval;
    }

    Point() { }  // reinstates the default no-args constructor

}
```

We have not so far written any interesting programs using classes. This is because interesting programs that use classes nearly always employ dynamic methods which we haven't yet studied.