Classes

In both Java and C++, the `class` is the primary implementation mechanism for ADTs. While there are a few syntactical differences, the basic definition, implementation, and use of `class`es is very similar in the two languages. Consider the following simple example (ignore for now the use of "`->`" versus "." in various places in the C++ version):

The definition of classes in C++ is somewhat different than in Java. Here is an example: a C++ version of the Point class:

```
class Point /* C++ */
{
public:
   Point();
   Point(double xval, double yval);
   void move(double dx, double dy);
   double getX() const;
   double getY() const;
private:
   double x;
   double y;
};
```

There are several essential differences.

1. In C++, there are public and private sections, started by the keywords public and private. In Java, each individual item must be tagged with public or private.

2. The class definition only contains the declarations of the methods. The actual implementations are listed separately.

3. Accessor methods are tagged with the keyword const

4. There is a semicolon at the end of the class

The implementation of methods follows the class definition. Because the methods are defined outside the classes, each method name is prefixed by the class name. The :: operator separates class and method name. Accessor methods that do not modify the implicit parameter are tagged as const.

```
Point::Point() { x = 0; y = 0; }

void Point::move(double dx, double dy)
{  x = x + dx;
   y = y + dy;
}

double Point::getX() const
{  return x;
}
```

Objects

The major difference between Java and C++ is the behavior of object variables. In C++, object variables hold values, not object references. Note that the new operator is never used when constructing objects in C++. You simply supply the construction parameters after the variable name.

```
Point p(1, 2); /* construct p */
```

If you do not supply construction parameters, then the object is constructed with the default constructor.

Time now; /* construct now with Time::Time() */

This is very different from Java. In Java, this command would merely create an uninitialized reference. In C++, it constructs an actual object.

When one object is assigned to another, a copy of the actual values is made. In Java, copying an object variable merely establishes a second reference to the object. Copying a C++ object is just like calling clone in Java. Modifying the copy does not change the original.

```
Point q = p; /* copies p into q */

q.move(1, 1); /* moves q but not p */
```

In most cases, the fact that objects behave like values is very convenient. There are, however, a number of situations where this behavior is undesirable.

1. When modifying an object in a function, you must remember to use call by reference (see below)

2. Two object variables cannot jointly access one object. If you need this effect in C++, then you need to use pointers.

3. An object variable can only hold values of a particular type. If you want a variable to hold objects from different subclasses, you need to use pointers

4. If you want a variable point to either null or to an actual object, then you need to use pointers in C++

Java	C++
`public class Complex`	`class Complex`
`{`	`{`
	`private:`
`    private double realPart,`	`        double realPart,`
`imaginaryPart;`	`imaginaryPart;`
	`public:`
`    public Complex()`	`    Complex()`
`    {`	`    {`
`        this.realPart = 0.0;`	`        this->realPart = 0.0;`
`        this.imaginaryPart = 0.0;`	`        this->imaginaryPart = 0.0;`
`    }`	`    }`
`    public Complex(double re, double`	`    Complex(double re, double im)`
`im)`	`    {`
`    {`	`        this->realPart = re;`
`        this.realPart = re;`	`        this->imaginaryPart = im;`
`        this.imaginaryPart = im;`	`    }`
`    }`	
	`    Complex add(Complex rhs)`
`    public Complex add(Complex rhs)`	`    {`
`    {`	`        return Complex(this-`
`        return new`	`>realPart+rhs.realPart,`
`Complex(this.realPart+rhs.realPart,`	`             this-`
	`>imaginaryPart+rhs.imaginaryPart);`
`this.imaginaryPart+rhs.imaginaryPart);`	`    }`
`    }`	
	`    double length()`
`    public double length()`	`    {`
`    {`	`        return sqrt(realPart*realPart`
`        return`	`+`
`Math.sqrt(realPart*realPart +`	
	`        imaginaryPart*imaginaryPart);`
`        imaginaryPart*imaginaryPart);`	`    }`

```
    }
}
```

```
};
```

The previous C++ example was constructed to look as similar as possible to the Java version. While perfectly valid, it is more conventional to split the definition and implementation of C++ classes into two files (see also **Separate compilation**). Hence the following is more representative of a typical C++ implementation:

`Complex.h`

```
class Complex
{
private:
        double realPart, imaginaryPart;

public:
    Complex();
    Complex(double re, double im);
    Complex add(Complex rhs);
    double length();
};
```

`Complex.c++`

```
#include "Complex.h"

Complex::Complex() : realPart(0.0), imaginaryPart(0.0)
{
}
Complex::Complex(double re, double im) : realPart(re), imaginaryPart(im)
{
}
Complex Complex::add(Complex rhs)
{
    return Complex(this->realPart+rhs.realPart,
        this->imaginaryPart+rhs.imaginaryPart);
}
double Complex::length()
{
    return sqrt(realPart*realPart +
        imaginaryPart*imaginaryPart);
}
```

Instances of `class Complex` can be created and used in a similar manner in the two languages:

Java

```
void client()
{
    Complex c1 = new Complex(1.8, -3.1);
    Complex c2 = new Complex(-1.1, 9.8);
    Complex c3 = c1.add(c2);
    …
}
```

C++: Version 2 of `client`

```
void client()
{
    Complex c1(1.8, -3.1);
    Complex c2(-1.1, 9.8);
    Complex c3 = c1.add(c2);
    …
}
```

Copy Constructor

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
The copy constructor is used to – Initialize one object from another of the same type. Copy an object to pass it as an argument to a function.

```
#include<iostream>
using namespace std;

class Point
{
private:
	int x, y;
public:
	Point(int x1, int y1) { x = x1; y = y1; }

	// Copy constructor
//	Point(const Point &p2) {x = p2.x; y = p2.y; }
	void setX(int a){ x= a;}

	int getX()			{ return x; }
	int getY()			{ return y; }
};

int main()
{
	Point p1(10, 15); // Normal constructor is called here
	Point p2 = p1; // Copy constructor is called here or Point p2(p1);

	// Let us access values assigned by constructors
	cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
	cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY()<<endl;
	p1.setX(4);
	cout<<p1.getX()<<endl;
	cout<<p2.getX()<<endl;

	return 0;
}
```

Output

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15

4
10
```