

Revision: objects

- Object in Java provides an data analog to methods:
 - A method call groups some statements together and say that it is one thing to do. A method defines what it really does.
 - An object groups some variables together and say it is a one piece of data to store. A class defines what it really stores.
- To support the idea that somebody creating an object need not know how it is implemented:
 - Instance methods define the interface for accessing the object.
 - User won't initialize an object, constructors will.
 - Other than the interface, everything is `private` to prevent the main program from incorrectly corrupting the object.

0911B L23-1

The concept of sub-type

- As said in previous lectures, objects are usually used to represent real-world or abstract concepts.
- Real world objects usually belong to multiple “types”. If I say “I want some fruit” (or “I want some food”), you can give me an apple, an orange or a banana and I should be happy with it.
- Of course, a particular apple is of “type” apple, but it is also of type fruit. In fact, *all instance of type-apple objects are also of type fruit*.
- We say that apple is a sub-type of fruit: it is a specialization of fruit. Apple is not fruit, but if somebody asks for fruit he will be happy if we give apple.
- Conversely, we say fruit is a super-type of apple.

0911B L23-2

Sub-types in Java

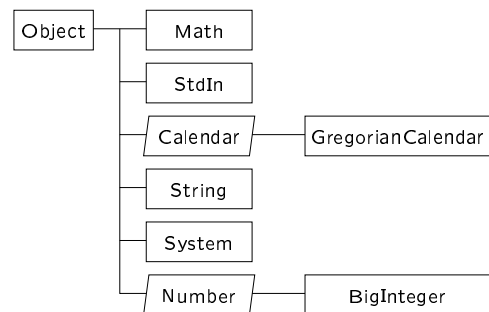
The designers of Java adopt a very simple version of sub-types.

- Every class has exactly one immediate super-class—no more, no less. In particular, it cannot has two super-classes. (Unnatural.)
- The super-class of a super-class is also a super-class.
- This rule (every class has an immediate super-class) holds for all classes except the ultimate super-class—Object.
- Every class is directly or indirectly a sub-class of Object.
- Even array types are sub-class of Object, so you can say arrays are actually objects.

0911B L23-3

The class hierarchy

Since all class has a super-class and the ultimate super-class is Object, the system can be depicted visually as a class hierarchy.



Note that some of the classes here are not usually instantiated (e.g., System or Math). But Java does not really distinguish them.

0911B L23-4

What it really means?

The question remains: what it really means to be a sub-class of another class? Suppose A is a subclass of B.

- A provides all the services that is provided by B. We say that A inherits all the methods of B.
- Looking in another way, all the promises of B are honored by A.
- Therefore, it is safe to use a type-A value at any place that want a type-B value.
- A type-B variable can hold a type-B value, so it can also hold a type-A value.
- But a type-A variable cannot hold a type-B value: a type-A variable want type-A values, not type-B values.

0911B L23-5

Example: shapes

For example, we might want a class to represent shapes. For all shapes, we want to be able to find their area, give a String representation of the shape, and enlarge it by a particular fraction.

Clearly, constructing a rectangle will be very different from constructing a circle. And finding the area of a rectangle will be very different from finding the area of a circle. But given any shape object, we want to be able to find their area in exactly the same way.

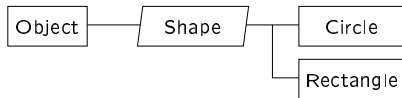
It should be possible to build an array containing shape-objects, where each of the elements is of a different shape.

And it should be possible to write a function taking an array of shapes and find the area of each of the shapes.

0911B L23-6

Designing our class hierarchy

Let's have two types of shapes, Circle and Rectangle. We make Shape a super-class of Circle and Rectangle. This results in the following class hierarchy.



Now it is a good point to explain what is the difference between the class name within rectangles and the class name within slanted parallelograms.

Those within parallelograms are not supposed to be instantiated. That is, there shouldn't be any Object of type Shape. We call such types "abstract types" in Java.

0911B L23-7

Why is it there?!

If we do not expect to have any objects of Shape, why we bother have it in the class hierarchy?

It is there for quite a few purpose:

- It gives a way for people to say "any shape will do, whatever shape". For example, if a method takes a Shape argument, we can give it a Rectangle or Circle.
- It defines what we can expect from a Shape. It may define that "any sub-class of shape will provide a way to find the area of such objects."
- It may provide default implementation of its sub-class. It may even say that "all sub-class must use this implementation".

0911B L23-8

Creating an abstract class

To create an abstract class, all we need to do is to mark the class **abstract**. If it wants to give a default implementation for a method, write it. Otherwise, just mark the methods as **abstract**.

```
abstract public class Shape {
    public abstract double getArea();
    public abstract void scale(double ratio);
    public abstract String getType();
    public String toString() {
        return getType();
    }
}
```

Since we don't know kind of shape is it, we don't know how to find the area, how to scale it and what to return as its own type.

But we can give a default implementation of toString: return whatever returned by getType.

0911B L23-9

Using Shape

Given such a Shape definition, we cannot create a Shape. But we can write methods like the following:

```
public class TestShape {
    ...
    // Scale all shapes in an array
    static void scaleShapes(Shape[] arr, double ratio) {
        for (int i = 0; i < arr.length; ++i)
            arr[i].scale(ratio);
    }

    // Print the shapes and the areas
    static void printShapesAndAreas(Shape[] arr) {
        for (int i = 0; i < arr.length; ++i) {
            System.out.print(arr[i].toString() + ": ");
            System.out.println("Area = " + arr[i].getArea());
        }
    }
}
```

Note that Shape provide a way for us to say "Shape, whatever the real type".

0911B L23-10

Making sub-class

There is nearly nothing new in defining sub-class. The only thing new is a new keyword, **extends**, meaning I want it to be a sub-class of something.

```
public class Circle extends Shape {
    public Circle(double r) { // Create a circle
        radius = r;
    }
    public double getArea() { // Get its area
        return radius * radius * Math.PI;
    }
    public void scale(double ratio) { // Scale it by a ratio
        radius *= ratio;
    }
    public String getType() { // Always return "Circle"
        return "Circle";
    }
    public String toString() { // Turn it to a string
        return "Circle(" + radius + ")";
    }
    private double radius;
}
```

Since all the abstract methods of Shape are implemented, now you can really make an object of Circle.

0911B L23-11

One more sub-class

```
public class Rectangle extends Shape {
    public Rectangle(double w, double h) { // Create a rectangle
        width = w;
        height = h;
    }
    public double getArea() { // Find its area
        return width * height;
    }
    public void scale(double ratio) { // Scale it by ratio
        width *= ratio;
        height *= ratio;
    }
    public void scaleX(double ratio) { // Scale horizontally
        width *= ratio;
    }
    public void scaleY(double ratio) { // Scale vertically
        height *= ratio;
    }
    public String getType() {
        return "Rectangle";
    }
    private double width, height;
}
```

Note that a class can contain more methods than its super-class.

0911B L23-12

Creating an array

Now we can really create some shapes. For example:

```

public class TestShape {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Rectangle(2, 3);
        shapes[1] = new Circle(5);
        shapes[2] = new Rectangle(1, 1);
        scaleShapes(shapes, 2.5);
        printShapesAndAreas(shapes);
    }
    ...
}

```

Note that shapes is an array of Shape references, but a Shape reference can refer to Rectangle and Circle objects.

On the other hand, we cannot write `shapes[0].scaleX(0.5)`: `scaleX` is not defined for Shape objects. It does not matter that it is really a Rectangle object or not.

0911B L23-13

What if I do want to treat it as such an object?

Now we have a problem: we know that `shapes[0]` is a Rectangle (we created it that way), but we cannot call a Rectangle method for it!

To do that, we do casting. For example, we can write:

```
((Rectangle)shapes[0]).scaleX(0.5);
```

If `shapes[0]` is really a Rectangle, it will call the `scaleX` method.

What if it is not a Rectangle? Then we get a run-time error. Can we check whether it will cause a run-time error before doing a cast?

Yes. The operator `instanceof` provide that capability.

0911B L23-14

Using instanceof

- Given a reference `obj` to object, we can say

```
obj instanceof Rectangle
```

to check whether it is of type Rectangle.

- For example, the following method scales all the rectangles horizontally by ratio:

```

static void scaleRectsHoriz(Shape[] arr, double ratio) {
    for (int i = 0; i < arr.length; ++i)
        if (arr[i] instanceof Rectangle)
            ((Rectangle)arr[i]).scaleX(ratio);
}

```

0911B L23-15

Full precedence table

Whenever you hear the word "operator", you're supposed to think about "precedence".

Operators	Associativity
<code>., method (), [], postfix ++, postfix --</code>	left
<code>unary +, unary -, !, prefix ++, prefix --</code>	right
<code>new, casting ()</code>	right
<code>*, /, %</code>	left
<code>binary +, binary -</code>	left
<code><, <=, >, >=, instanceof</code>	left
<code>==, !=</code>	left
<code>&</code>	left
<code>^</code>	left
<code> </code>	left
<code>&&</code>	left
<code> </code>	left
<code>?:</code>	right
<code>=, +=, -=, *=, /=, %=</code>	right

0911B L23-16

Class announcement

- Don't forget to hand in assignments by next Monday!
- Last workshop: Dec 5–6, 2000: a glimpse at GUI programming.
- Exam date: finalized. 1430–1730, Jan 3, 2000.
- You may take one piece of A4 paper containing any written/printed material with you during exam. Be wise when you prepare it!
- Exam appendix: the examination paper contains 3 tables: precedence table, keyword table and data range table. Don't waste the space of your A4 paper for those purposes!
- The exam contains 15 short questions, 1 program to debug, and 3 methods/programs to write. That's all about the exam.

0911B L23-17