

Java Basics

1. INHERITANCE

Code sharing is a good thing. First, you do not want to re-write the same stuff more than once. Also, code is likely to be more reliable as it only has to be debugged once. A mechanism for sharing code is *inheritance*.

```
class A {
    //variables
    //methods
}
class B extends A {
    //class B inherits ALL variables and methods of class A
}
```

We say that class A is the parent of class B, or the *super-class* of class B; class B is the *sub-class*. A class can inherit from a single other class (in Java).

The sub-class inherits all the variables and methods in the super-class. It is as though the variables and methods of the super-class would be textually present in the sub-class.

Example:

```
class OneDPoint

class TwoDPoint extends OneDPoint

class ThreeDPoint extends TwoDPoint
```

Example:

```
class Person {
    String name; //the name

    void Person(String s) {
        name = new String(s);
    }

    String getName() {
        return name;
    }
};
class Student extends Person {

    int class; //the class of graduation
```

```

    void Student(String n, int year) {
        super(n);
        class = year;
    }
}

```

Now we can say:

```

class Student s = new Student("Jon Myers");
System.out.println("name is " + s.getName());

```

In the constructor of the sub-class, the first thing to do is invoke the constructor of the super-class.

In our toy example above this makes no sense because class `Person` is empty and it does not even have a constructor. But in general it takes some work to “construct” a parent class, so in a sub-class we first need to invoke the appropriate constructor and pass it the arguments that we want.

Object-oriented programming allows to define class hierarchies. That is, it allows for classes to inherit from other classes, which inherit from other classes, and so on.

1.1 Overriding methods

We said earlier that a sub-class inherits all the methods of the parent classes. Sometimes it is the case that a sub-class is not happy with the definition of a method inherited from the parent; it might want to do something different. In this case it may choose to redefine the method. This is called “overriding a method”.

1.2 The Java class hierarchy

In Java the root of all classes is class `Object`. That is, every Java class inherits from it.

Class `Object` has no variables, but has a bunch of methods. Thus any class will inherit these methods, and their default definition in class `Object`. Here are a few that we’ll be using later:

```

//Indicates whether some other object is "equal to" this one.
boolean equals(Object obj);

//Returns a string representation of the object.
String toString()

```

2. JAVA INTERFACES

Java has the concept of an **interface**. An interface is like a class, just that it does not give the implementation, just the *signatures* of the methods. All methods in an interface must be public.

```
public interface xxx {

    // list the signatures of the methods
}
```

For example, here is an interface that you'll use all the time:

```
public interface MouseListener {
    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

Interfaces are used to describe a class *without* an implementation.

A class may choose to *implement* an interface, or several. Every class that implements an interface must define all methods in the interface.

When compiling a class that implements an interface, the compiler checks to see that *all* methods specified in the interface are implemented; the parameter types and return type must match; if not, it gives an error. So you can think of an interface as a way of specifying a set of methods that an object must support. An interface is a contract between a class and the outside world.

It is possible that a class implements several interfaces—then it needs to implement *all* methods in *all* interfaces.

You probably won't be defining interfaces, but you'll write all the time classes that implement interfaces.

Interfaces are useful because sometimes you don't know exactly what class you'll be using, you just know it has to have some functionality. You can think of interfaces as *generic* classes. For example, here is the set of methods that we'd expect from a `List`:

```
public interface List {

    //return the size of the list
    public int size();

    //return true if list is empty
    public boolean isEmpty();

    //return true if the list contains the given element
    public boolean contains(Object value);

    //add one element to the list
```

```

    public void add(Object value);

    //remove the element from the list
    public void remove(Object value);
}

```

Suppose we have two classes that implement `List`

```

public class ArrayList implements List {

    //... implements all methods in the interface, and possibly more
}

public class Vector implements List {

    //... implements all methods in the interface, and possibly more
}

```

Now we can write a piece of code that works with a generic `List`:

```

void collectData(List l) {
    ....
}

```

and we can call this passing an argument either an `ArrayList` or a `Vector`.

3. INHERITANCE AND DYNAMIC TYPE-CASTING

Java performs automatic type conversion from a sub-type to a super-type. That is, if a method requires a parameter of type `A`, we can call the method with a parameter that is a sub-class of `A`. Basically this says that any class that inherits from `A` can be used in place of type `A`.

One way you will see this is when writing, say, a class that is supposed to handle an array of objects— but we don't know what the type of the objects will be. So we declare the data as:

```

class Vector {
    Object[] myArray;
    ...

    //insert x in the array
    void insert(Object x) {...}
}

```

Then we can use `Vector` to hold *any* objects.

```

Vector v = new Vector();
v.insert('snow');
v.insert('cold');

```

4. METHODS INHERITED FROM JAVA CLASS OBJECT

4.1 toString()

This is a method that returns the string representation of the object its called on. It is defined in class `Object` and therefore inherited by all objects.

For example,

```
class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}

public class ToStringDemo {
    public static void main(String args[]) {
        Point p = new Point(10, 10);

        // using the Default Object.toString() Method
        System.out.println("Object toString() method : "+p);

        // implicitly call toString()
        String s = p + " testing";
        System.out.println(s);
    }
}
```

Try playing with this.

If you want more useful information about an object (for e.g. helpful in debugging) you'll have to override the `toString` method:

```
public void toString() {
    System.out.println("x=" + x + ", y=" + y);
}
```

4.2 equals vs. ==

You have seen `==` on integers, and other primitive types. You can also use `==` on objects. There is a difference though: on objects `==` determines whether two objects *refer to the same object*. That is, `==` compares *reference*, not values.

```
SomeClass x, y, z;
```

```
x = new SomeClass();
y = x;
//then x==y will return true
```

```
z = a copy of x
//then x==z will return false, even though the values are equal
```

What if we want to check whether the two objects have the same fields? Any class has a (default) `equals()` method that is inherited from class `Object`. The default version of `equals()` checks for equality of reference, just like `==`.

Usually you are not interested whether the two objects refer to the same object, but rather, whether two objects are equal—i.e. their fields are equal. If you want to check data equality, you need to implement (override) .

This has been done for example on class `String`. The `equals` method on class `String` checks for data equality. For example,

```
String s = "Bowdoin";
if (s.equals("bowdoin"))
```

will return `true`.

5. STATIC VARIABLES AND METHODS

When a number of objects are created from the same class, they each have their own distinct copies of instance variables, stored in different memory locations.

A class can have variables that are declared `static`. These variables are associated with the *class*, not with the instances of the class (i.e. the objects).

The static variables are common to the class and are shared by all instances of the class. Any object can change the value of a class variable; class variables can also be manipulated without creating an instance of the class.

A static variable is accessed as `<class-name>.<variable-name>`.

Example: a `BankAccount` class can keep a static variable for the number of accounts open. Every call to a constructor would increment this variable.

A common use of static variables is to define "constants". Examples from the Java library are `Math.PI` or `Color.RED`.

A method can also be declared `static`. Then it is associated with the class, and not with an object. From inside a class it is called by its name, just like a non-static method. From outside the class it is called on the class `<class-name>.<method-name>(<arguments>)`.

You may ask: when to declare a method static? When it performs a function specific to the class, not to a specific object.

```
public class Math {
    ...
    //returns true if n is prime
    public static boolean isPrime(int n) {
        ...
    }
}
```

To find whether a number is prime, you don't need to create a specific object. From outside the class, call `Math.isPrime(645)`.

A common use for static methods is to access static variables.

The functions in `Math` are all static: `Math.sin(x)`, `Math.cos(x)`, `Math.exp(d)`, `math.pow(d1, d2)`.

Note: static methods cannot access instance variables (there is no instance), and cannot access `this` (there is no instance for `this` to refer to).

6. SCOPE

Types of variables in Java:

- (1) instance variables (non-static)
- (2) class variables (static variables)
- (3) local variables
- (4) parameters

Scope: Within a method. Between methods. Between classes.

Variables must be declared within a class (instance variables or global variables) or a method (local variables).

An instance variable is visible/accessible to all methods of that class. If it is public, it is also visible outside the class.

A local variable is visible only inside the method; in the code following its declaration, within the tightest enclosing .

The parameters of a method are visible (only) inside the method and do not need to be re-declared inside.

6.1 `this`

Used to refer to an instance variable that clashes with a parameter:

```
class A {
    private int age;

    public void setAge(int age) {
        this.age = age;
    }
}
```

6.2 Example

```
public class ScopeTester {

    //instance variable
    private int x = 1;

    public void test1 (int x) {
        //a parameter shadows an instance variable
        System.out.println("x = " + x);
        x = 10;
        System.out.println("x = " + x);
        System.out.println("this.x = " + this.x);
    }
}
```

```

        public void test2() {
//you can redeclare an instance variable
double x = 10.3;
System.out.println("x=" + x);
System.out.println("this.x=" + this.x);
        }

        public void print() {
System.out.println("x=" + x);
        }

        public boolean equals (ScopeTester s) {
return (x == s.x);
        }

        public static void main (String args[]) {
ScopeTester s = new ScopeTester();

int a = 100;
System.out.println("before test1: a = " + a);
s.test1(a);
//a is changed in test1(), but not visible outside
System.out.println("after test1: a = " + a);

s.test2();
s.print();

ScopeTester x, y;
x = new ScopeTester();
y = new ScopeTester();
if (x==y) System.out.println("same object");
else System.out.println("NOT same object");
if (x.equals(y)) System.out.println("equal");
else System.out.println("not equal");

System.out.println("this is the default toString(): " + x);

        }
}

```

7. PROGRAMMING WITH ASSERTIONS

A good programming practice in any language is to use assertions in the code to enforce invariants.

For example, if at some point in a method we know that a variable say *x* must be non-zero, we write:


```
assert (x != 0);
```

When the code is executed, if x is non-zero, the assertion succeeds. Otherwise, if x is zero, the assertion fails, and the program terminates with an error.

In order to enable assertion you have to compile with the flag `-ea`:

```
javac -ea xxx.java
```

Assertions don't have any effect on the code as such, they are just used to make sure that certain conditions are true. It is good practice and style to include assertions in your code. They help you catch errors earlier.

8. TESTING AND DEBUGGING

If your program compiles, that does not mean it is correct. It can still have bugs... that is, logical errors. You will find that you will spend most of your time debugging your code. There is no recipe on how to debug — just plenty of print statements, and/or use BlueJ.

To find all possible bugs, you need to *test* your program on different inputs, and take into account all situations that may appear.