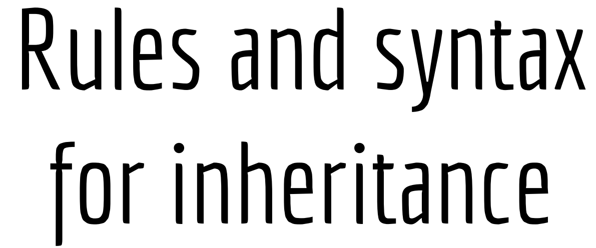

Rules and syntax for inheritance

The boring stuff

The compiler adds a call to super()

Unless you explicitly call the constructor of the superclass, using `super()`, the compiler will add such a call for you *as the first instruction of your constructor*.

If you want to call a constructor of the superclass explicitly, it must be done as the first instruction of the constructor.

The implicit call is of course to the no-arguments-constructor of the superclass, so if your superclass lacks such a constructor, you must call one of the existing constructors yourself, or you will have a compiler error.

Do not add a no-arguments-constructor just to fix this “problem”!

You can only extend one class (at the time)

Java uses a single-rooted hierarchy for inheritance, which means that you can only extend one class directly.

There is another construct for achieving something similar to declaring a class as a subtype to many types, which is called interfaces. We'll look at interfaces in a later chapter.

You can prevent your class from being extended

If you want to say that your class cannot be (and should not be) extended, you can use the keyword `final` at the class declaration:

```
public final class String /*this class cannot be extended*/
```

Unless you have a very good reason to let people extend your classes, we recommend that you use the final keyword to prevent inheritance.

The reason is that it is very complicated to design a class meant for inheritance and avoid the pitfalls which follow with inheritance.

Another way to prevent your class from extension

You can also make all constructors private in order to prevent people from extending your class.

Instead of public constructors, you can write public static methods which create and return new instances of your class.

Such methods are often called factories.

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();
    public static Runtime getRuntime() {
        return currentRuntime;
    }
    private Runtime() {} // no one can make an instance (or inherit)
...
```

Sometimes you want to keep some methods

To prevent a single method from being overridden (re-defined) in a subclass, you can also use the `final` keyword.

A final method cannot be overridden.

Methods called from the constructor must be final, because if they are overridden, they will be called before the subclass has been instantiated.

Non-final method call FAIL

```
public class Super{

  public Super(){
    method();// uh-oh
  }
  public void method(){
  }
}
class Sub extends Super{
  private String s;
  public Sub(){
    s="abc"; // too
late!
  }
  @Override
  public void method(){
    s=s.toUpperCase();
  }
}
```

```
public class TestSub{
  public static void main(String[] args){
    Sub sub = new Sub();
  }
}
```

```
$ java TestSub
Exception in thread "main"
java.lang.NullPointerException
    at Sub.method(Super.java:21)
    at Super.<init>(Super.java:4)
    at Sub.<init>(Super.java:16)
    at TestSub.main(Super.java:11)
```

What just happened?

```
class Sub extends Super{
  private String s;
  public Sub(){
    s="abc"; // too late!
  }
  @Override
  public void method(){
    s=s.toUpperCase();
  }
}
```

```
public class Super{

  public Super(){
    method();// uh-oh
  }
  public void method(){
  }
}
```

Before s is initialized, an implicit call to super() is done. In the constructor of the super class we have: `method();`

But method is overriden, so the overridden version is called. S is still null!

Checking class type at runtime

The `instanceof` operator checks class affiliation.

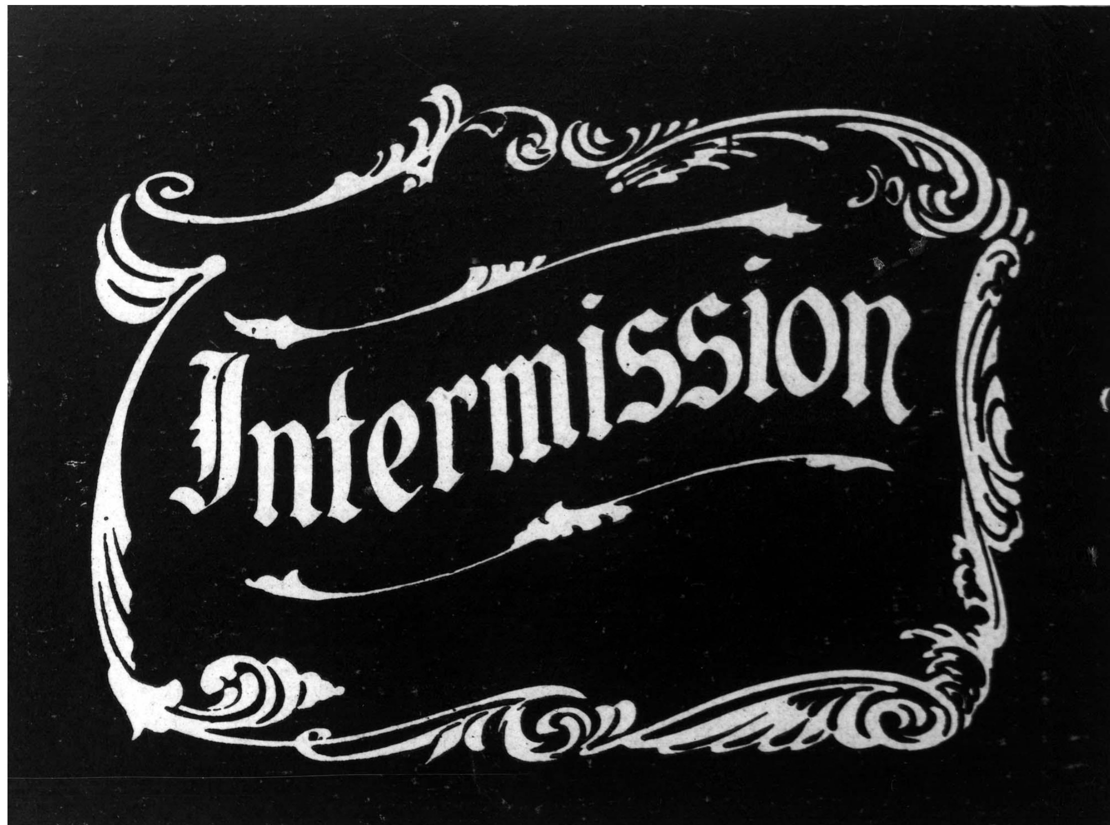
It checks if the left hand operand is an instance of the right hand operand or any of its subclasses. Special case: null is not an instance of anything.

```
String s = "";

Object o = new Object();

if(s instanceof Object){} // true, String extends Object

if(o instanceof String){} // false
```

Getting the actual class

You can use the getClass() method inherited from Object, in order to get a handle to the actual class of an object.

```
Object o = new String();
System.out.println("o.getClass().getName(): " + o.getClass().getName());
// java.lang.String will be printed
```



CC-BY City of Vancouver Archives - https://www.flickr.com/photos/vancouver-archives/8229393567

Anonymous class

You can create an extending class on the fly like this:

```
public class Anonymous{
  public static void main(String[] args){
    System.out.println( new Object(){
        public String toString(){
          return "I am an object";
        }
      });
  }
}
// Will extend Object and override toString().
// Will print "I am an object"
```

Collections of a Superclass

It is possible to create a collection object, like a list, with references to a supertype. While this is convenient, it is also pretty dangerous as we will show.

Let’s assume we have an array of Object references.

We know that each object reference in the array can refer to an object of any class type, since everything is an object (every class can be viewed as also being of type Object).

This might sound like a good idea first, since we don’t have to worry about whether we can assign the references in the array to some object - it will always work!

Collections of a Superclass

```
Object[] objs = new Object[4];
objs[0] = new String("ABC");
objs[1] = new Integer(4);
objs[2] = new Customer();
objs[3] = new Passport();

// Legal, since all classes are subtypes to Object
```

Collections of a Superclass

```
Object[] objs = new Object[4];
objs[0] = new String("ABC");
objs[1] = new Integer(4);
objs[2] = new Customer();
objs[3] = new Passport();
```

But is it good? The compiler and runtime let us put anything in the array. This is often not what we want. Imagine if we have an array presumably holding references to a list of, say, Customer objects. If we declare that array as above (of type array of Object references), nothing is preventing us or anyone else from populating the array with references to something else...

ArrayList example - First atempt

We could use an ArrayList instead for our customers:

```
ArrayList customers = new ArrayList();
```

But as soon as we add a reference to e.g. a Customer to the list, we'll get a warning from the compiler:

```
Note: SomeClass.java uses unchecked or unsafe operations.
```

The reason is that an ArrayList created as above can hold any type of references, just like the array of Object references before, which is not safe.

ArrayList example - First atempt - failure

Created like below, the list could hold references to any class.

```
ArrayList customers = new ArrayList();
```

So nothing is preventing us from also adding a String to the list, together with some Customer references.

When we get a reference from the list, like this: `customers.get(0);`

we will actually get the reference as of type Object. So we’d need to cast it to a Customer if that’s what we want. But if it’s really a String, that cast would fail:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to Customer
```

ArrayList example - Limiting type to Customer

Rather, we should declare the ArrayList as only accepting Customer references:

```
ArrayList<Customer> customers = new ArrayList<>();
```

That syntax uses a concept called “generics”. You may think of it as a typesafe ArrayList (or any collection class) which in the example above creates an ArrayList which will only accept references to Customers.

Not only that, when getting a reference from the ArrayList, it will automatically be of the correct type (Customer), whereas the bad ArrayList would only return references as Object references (which we would need to cast).