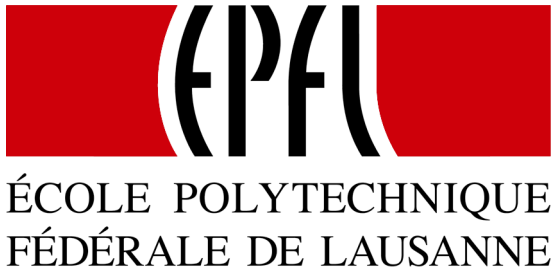


Java Refresher Course

Stanislav Chachkov

Department of Computer Science
Software Engineering Laboratory
November 2000

Contents

- Java Language Basics
- Core Java Libraries
- Building GUI with Swing
- Network Programming with Java Sockets and RMI

Useful Links

Main Java Site - http://www.javasoft.com

EPFL Java Site (docs and tutorial!) - http://sunwww:1995.epfl.ch/Java/

Java Developer Connecton (JDC!) - http://developer.java.sun.com/

"Examplets"! - http://developer.java.sun.com/developer/codesamples/examplets/

Bug Database - http://developer.java.sun.com/developer/bugParade/

Good book: "Java Threads (Java Series)"
by Scott Oaks, Henry Wong, Mike Loukides (Editor)
O'Reilly & Associates, 319 pages, 2nd edition (January 1999)

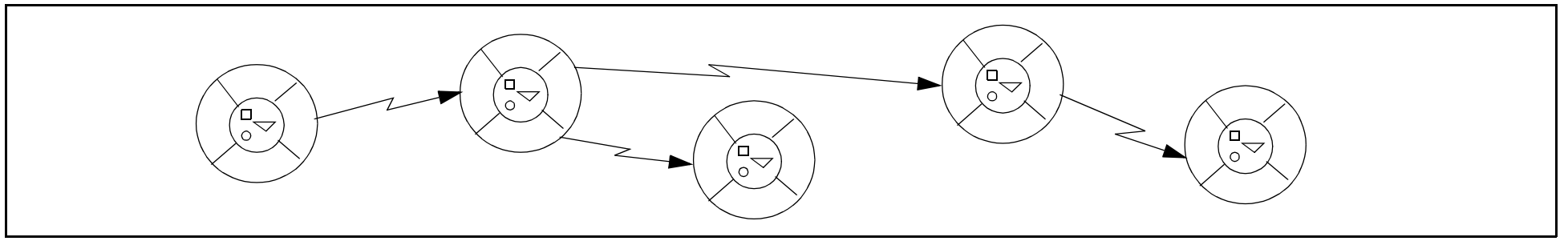
Java is:

- Object-Oriented ...
- Strongly Typed ...
- Interpreted ...
- Single-Inheritance ...
- Multi threaded ...
- Reference-based ...
- Memory-managed ...
- Case-sensitive ...

... programming language with a large quantity of libraries.

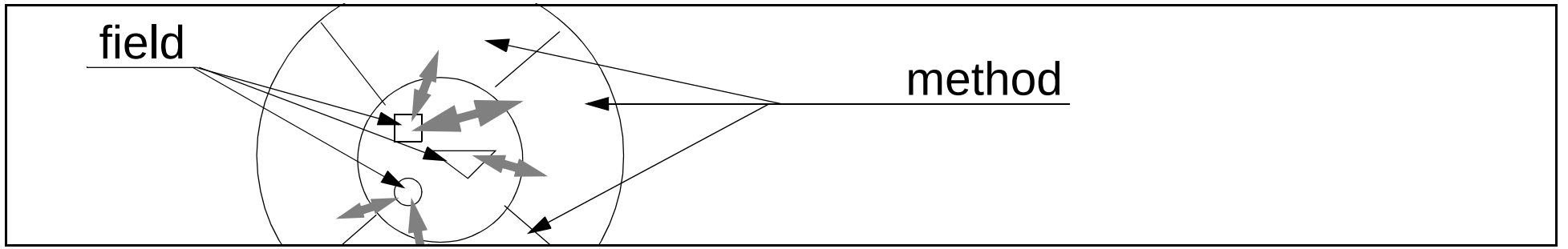
Objects and Classes

- A Java program is composed of communicating *Objects (and **nothing***



else)

- A Java *Object* is Composed of *Variables*(a.k.a. *Fields*) and *Methods*



- Object structure (i.e. code) is defined in object's *Class* (== type)
- Each object is an *instance* of some *Class*

Classes, Files and Packages

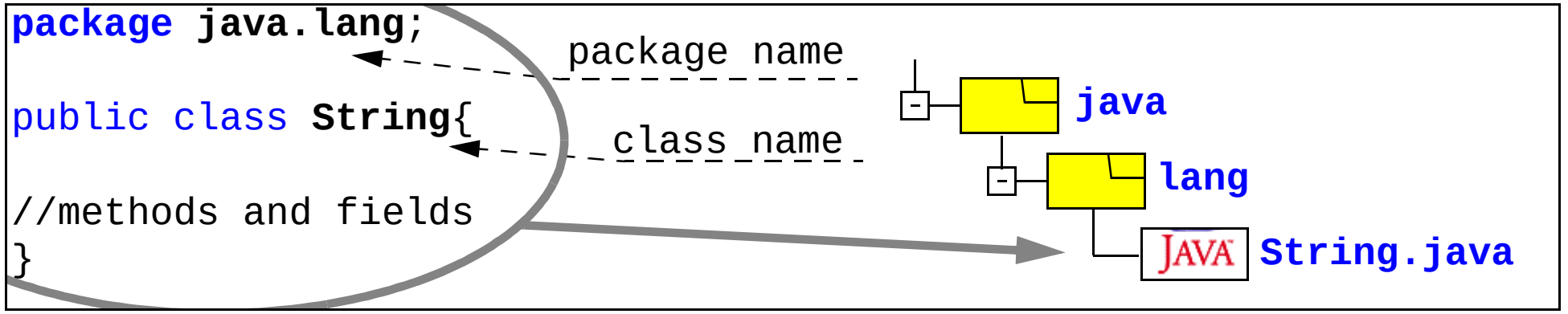
Java source code is stored in files with *.java* extension

- One Class == One File , File Name == Class Name*.java* (case sensitive!)

Classes can be grouped in hierarchical *packages*

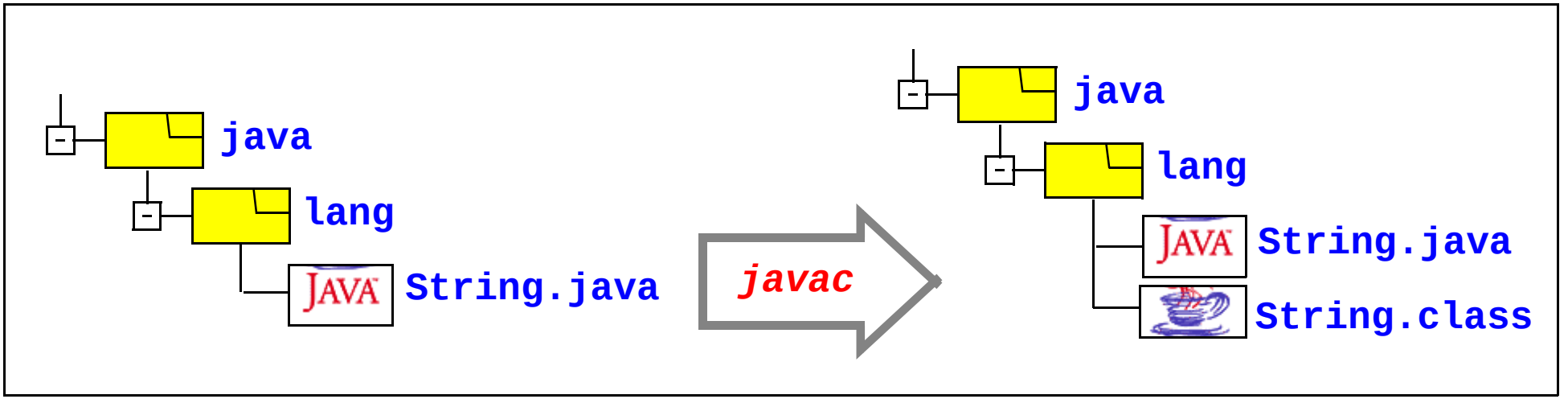
- One Package == One Directory, Package Name == Directory Name
- Sub-Package == Sub Directory

Full name of a class is: *packageName.ClassName*



Compile and Execute

- To be executed (interpreted) Java class have to be *compiled* first
- Compiled class is made of *bytecode*, stored in file with *.class* extension



- *Bytecode* == machine code for *Java Virtual Machine (JVM)*
- A compiled class can be **loaded** and executed by *JVM*
- There is **no** separated link edition phase in Java and **no** other file formats (e.g. *.obj*)

CLASSPATH and *ClassLoader*

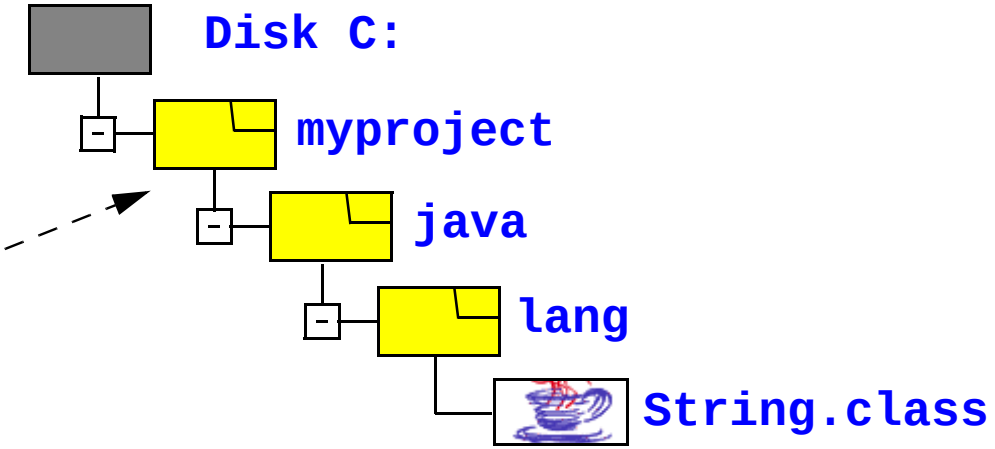
ClassLoader is the part of JVM responsible for loading classes

CLASSPATH is the list of places where classes can be found

ClassLoader uses *CLASSPATH* to find classes

Example: a CLASSPATH indicating where *java.lang.String* class can be found

```
set CLASSPATH=C:\myproject
```



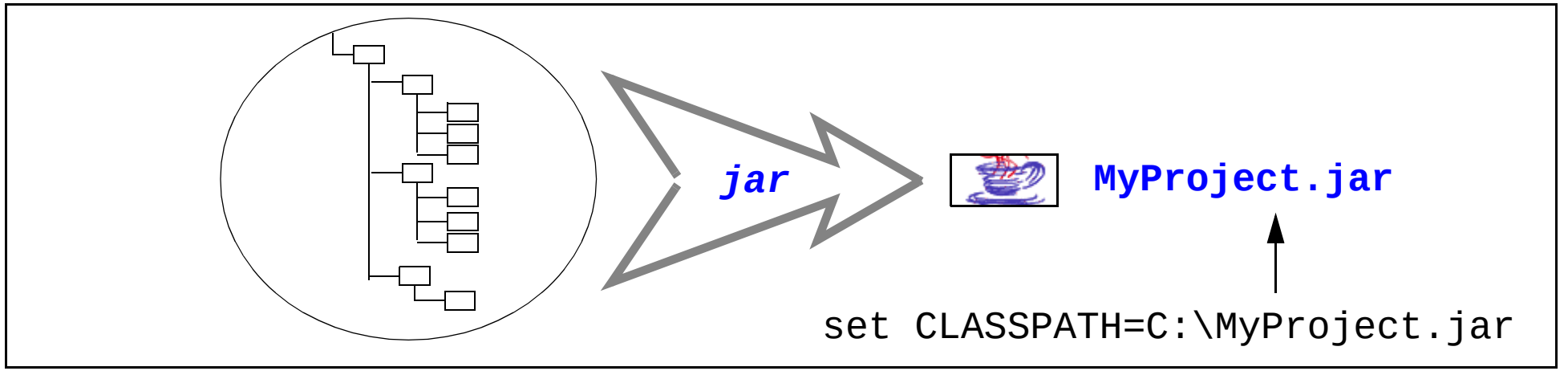
Example *UNIX*: `setenv CLASSPATH .:/home/user/myproject`

Example *Windows*: `setenv CLASSPATH=.;C:\myproject`

What's a JAR?

Problem : Large Java program -> A lot of files and folders ...

... Solution : Put'em all together!



JAR contains java classes but also **all other *resources*** that you need:

- Images, sounds, data etc.

JAR also contains a special file describing its contents:
`META-INF/Manifest.mf`

How to run Java Program

The entry point of a Java program is a method with the following signature:

```
public static void main(String[] argv)
```

Put this method in your program “main” class and you will be able launch it from the command line like this:

```
C:\> java myproject.MyMain
```

Example:

```
package myproject;
public class MyMain{
    public static void main(String[]argv){
        System.out.println(“Hello world!”);
    }
}
```

“Main” class must be **public**

Execute a JAR? Why not!

An *Executable JAR* can be launched either like this:

```
C:\> java -jar MyProject.jar
```

or by double-clicking on it in your file browser.

To create an Executable JAR :

1. Create a JAR with all your program classes and resources;

2. Add the “main” class name in `META-INF/Manifest.mf` like this:

```
Main-Class: myproject.MyMain
```

A Class: Let's look inside!

A class declares variables (*fields*) and *methods*. Example:

```
package examples;

public class Buddy{
      private String name;
      private Buddy friend = null;

      public Buddy(String name){
            this.name = name;
      }

      public String toString(){
            return "I am "+name;
      }

      public void setFriend(Buddy friend){
            this.friend = friend;
      }

      public Buddy getFriend(){
            return friend;
      }
}
```

Class name: Buddy
Full class name: examples.Buddy

Constructor: public Buddy(String name)

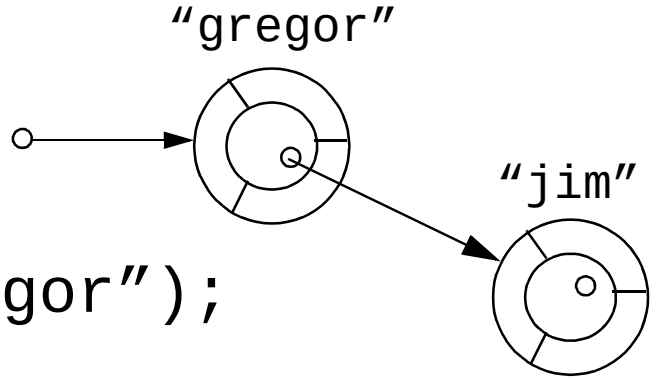
Methods:
public String toString()
public void setFriend(Buddy friend)
public Buddy getFriend()

Fields:
private String name
private Buddy friend

Constructors

Object instances are created using *Constructors.*

Example:




```
Buddy gregor = new Buddy(“Gregor”);

gregor.setFriend( new Buddy(“Jim”) );
```

Constructor is a special method that first **creates** (implicit) then **initializes** and **returns** (also implicit) a **new object**.

Constructor’s name == class name.

new keyword is used to invoke a constructor

Destructors

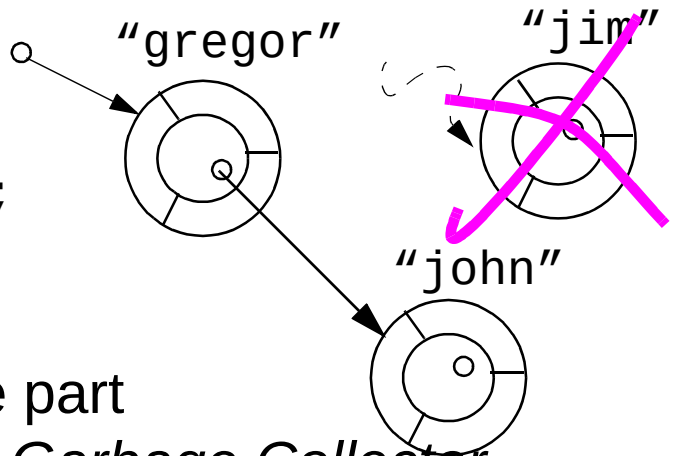
None

There is **nothing** in Java that corresponds to well-known (C++, Ada, ...) notion of *Destructor*.

You can **not** say “destroy yourself” to an object.

An object is automatically *destroyed* (deleted from memory) only when there’s no more references to it.

```
gregor.setFriend( new Buddy(“Jim”) );
gregor.setFriend( new Buddy(“John”) );
```

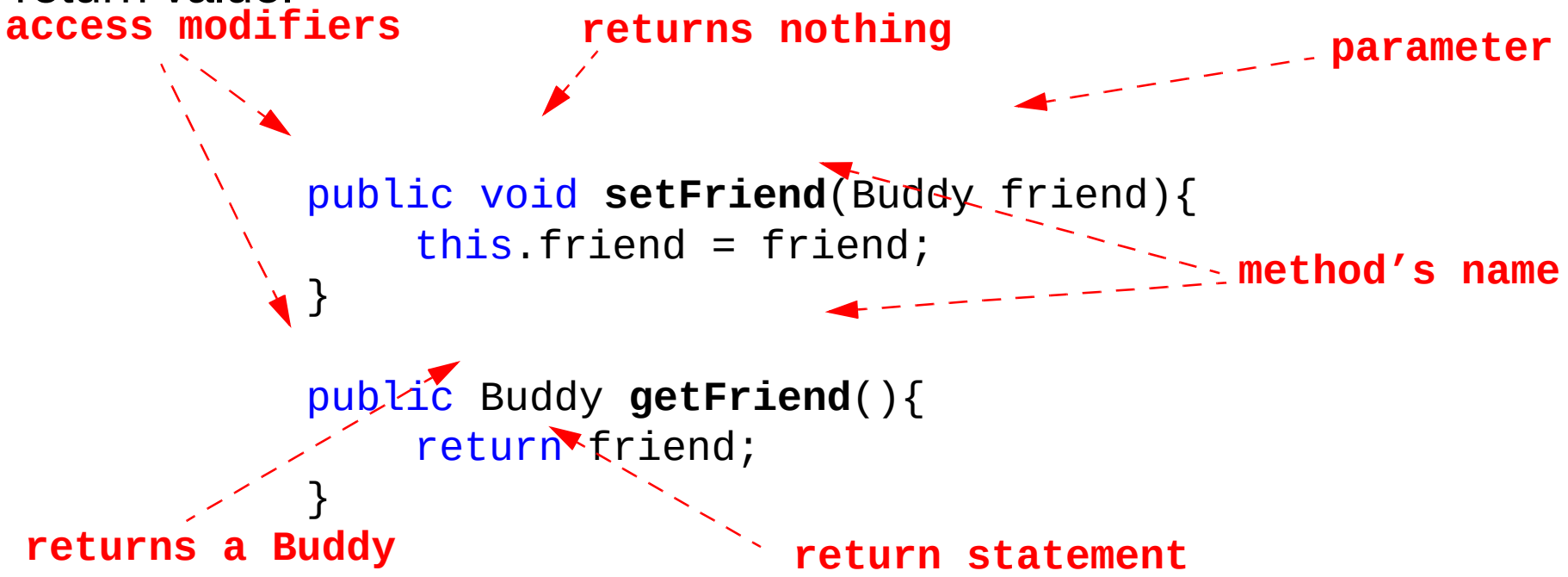


This process is called *Garbage Collection*. The part of the JVM that performs this task is called the *Garbage Collector*.

Methods

Methods define object behavior

- Contain executable code (instructions): read/write variables, call other methods, etc.
- Can take parameters (objects by reference, primitives by value) and can return value.



Fields

Fields store object's state (a.k.a instance variables)

- Fields are typed.
- Fields can be initialized or not. An uninitialized field is implicitly initialized to "zero" value: *null*, *0*, *false*, etc. depending on the field's type.

```
private String name; //implicitly =null
private Buddy friend = null; //explicitly =null
```

- Constants are declared like this:

```
public static final int MAX_VALUE = 2000;
```

Modifiers

Keywords in front of some members (method and field declarations):

Table 1: Visibility Modifiers

Modifier	Meaning
public	this member is visible from everywhere
protected	this member is visible only from subclasses and same package
private	this member is visible only from defining class
no modifier	a.k.a friend - this member is visible from same package only

and also some others:

- **static** - member is shared by all instances
- **final** - member that can not be redefined
- **abstract** - not fully defined method or class.

Types

In Java there is TWO kind of types:

- *Objects* - or references to objects, including arrays, “zero” value is called *null*

- Primitive types:

Table 2: Primitive Types

Type	Size or Values
int	32 bit, from -2147483648 to 2147483647
boolean	true or false
char	16 bit, from '\u0000' to '\uffff' inclusive, that is, from 0 to 65535
double	64 bit
float	32 bit
byte	8 bit, from -128 to 127
short	16 bit, from -32768 to 32767
long	64 bit, from -9223372036854775808 to 9223372036854775807

Arrays

Typed, Fixed-size, indexed by *int*, **0**-based

Multidimensional *arrays* are *arrays* of *arrays* (of *arrays* of ...)

Java arrays are **Objects**, but they cannot be sub-classed (*final*)

```
String[]a1; //implicitly =null
String a2[]; //other syntax, also =null
String a3[] = new String[12]; //array of 12 Strings, initialized to
                              //12 null references
String a4[][] = new String[2][2]; //array of 2 arrays of 2 Strings

a2 = new String[2]; //array of 2 Strings
a2[1]="B";
a3[0]=a2[1];
a4[1][0]="A";

Object arr = new String[3][2][1];//arrays are objects
```

Arrays: compact initialization

Arrays can be created and initialized in one expression

Useful for parameters, variable initializers etc.

```
a2 = new String[]{“A”,“B”};
a4 = new String[][] { {“A”,“B”} , {“C”, “D”} };
a4[1][0].equals( “C” )
```

Length

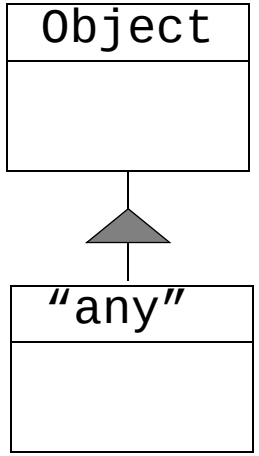
The length of an array is stored in public instance field *length*

```
a2.length == 2
a4.length == 2
a4[1].length == 2
```

Array index is in **[0..array.length)**

Inheritance

One root Class: ***java.lang.Object***




```
public class String extends java.lang.Object{...
==
public class String{... //default
```

No multiple inheritance allowed!

Super-class *methods* and *fields* access: *super* keyword

```
public void show(){
     super.show();
}
```

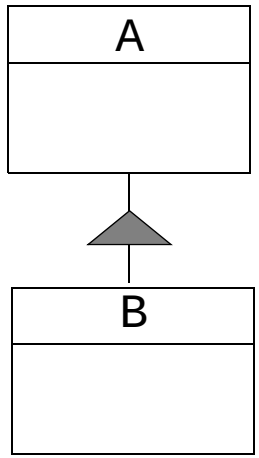
Not allowed:

```
super.super
```

Inheritance and Constructors

A constructor **must** first invoke a superclass constructor (any of)

```
public class A{ //extends java.lang.Object ;))
     public A(int i){}
}

public class B extends A{
     public B(){
          super(10);
     }
}
```



If no explicit *super* call found in constructor the compiler **insert** it:

`public B(){}` **is transformed in** `public B(){super();}`

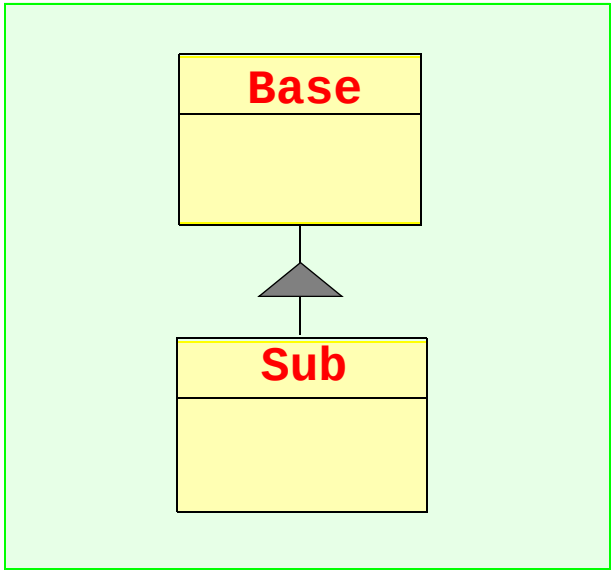
A problem? : `class A{ private A(){}}//don't subclass me ;(`

Inheritance by Example (I)

```
public class Base{
     protected int a;
     private int b;

     public void show(){}
     protected void me(){}
     private void more(){ fun(); } //c/t error
}

public class Sub extends Base{

     public void fun(){
          int x = a; //OK
          int y = b; //c/t error
          me(); //"Basic" me
     }
     public void show(){
          show();//recursive call
          super.show();//"Basic" show
          more();//c/t error
     }
}
```

Inheritance by Example (II)

```
public class Base{}
public class Sub extends Base{}
```




```
public class Other{
     public void m1(Sub s){
          m2(new Base()); //OK
          m2(new Sub());//OK
     }
     public void m2(Base s){
          m1(new Base()); //compile-time error
          m1(new Sub());//OK
          m1((Sub)new Base()); //run-time exception
```

Java Naming Style

"Case-sensitive". Easy. You must respect it!

```
public class BigCar{ // bad: bigcar, bigCar, big_car, BIGCAR, ...
     private int curSpeed; //bad: CurSpeed, cur_speed, CUR_SPEED,...
     public static final int NO_RULEZ=0; //constant

     public void goAndReturn(boolean noRoute){ //bad: _nO_rOuTe_
          ...
     }
}
```

- Lower-case in general
- First letter: capital for class names, small for other names
- Compound names: Each new word starts with a capital
- Constants: all letters capital, words separated by underscore

Why naming style? Example:

```
Car.stop() //looks like static method call
car.stop() //looks like instance method call

car.move(RIGHT_DIR) //parameter looks like constant
car.move(rightDir) //parameter looks like variable
```

Modifiers style:

Your “default” modifiers must be:

- public for methods and classes
- private for fields

Abstract methods and classes

abstract method is a method with **no body**.

```
public abstract void goSlowly( ); //no body!!!
```

Abstract methods can **not** be **invoked**!

```
car.goSlowly(); //compile-time error!
```

Class containing abstract methods **must** be declared abstract:

```
public abstract class Car{ //italics for abstracts, like in UML
    public abstract void goSlowly( ); //no body!!
}
```

Abstract classes can **not** be **instantiated**:

```
    new Car(); //compile-time error!
```

Class without abstract methods can also be declared abstract

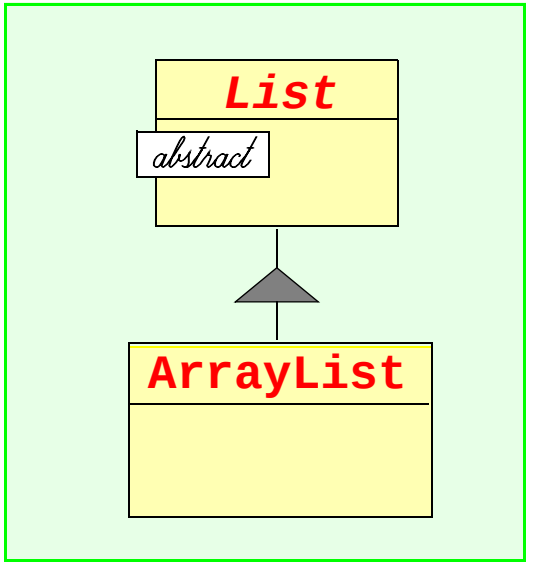
Abstract classes can have all kinds of members (*fields*, etc.)

Constructors and *fields* cannot be declared abstract

Why abstract methods? Example:

```
public abstract class List{
    public abstract void set(Object o,int index);
    public abstract Object get(int index);
    public abstract int size();
    public int find(Object o){
        for(int i=0;i<size();i++) if(o==get(i))return i;
        return null;
    }
    public void shuffle(){...}
    public void reverse(){...}
    public void sort(){...}
}

public class ArrayList extends List{
    /** concrete data storage*/
    private Object[] data;

    public void set(Object o,int index){ data[index]=object; }
    public Object get(int index){ return data[index]; }
    public int size() { return data.length; }
}
```

Use



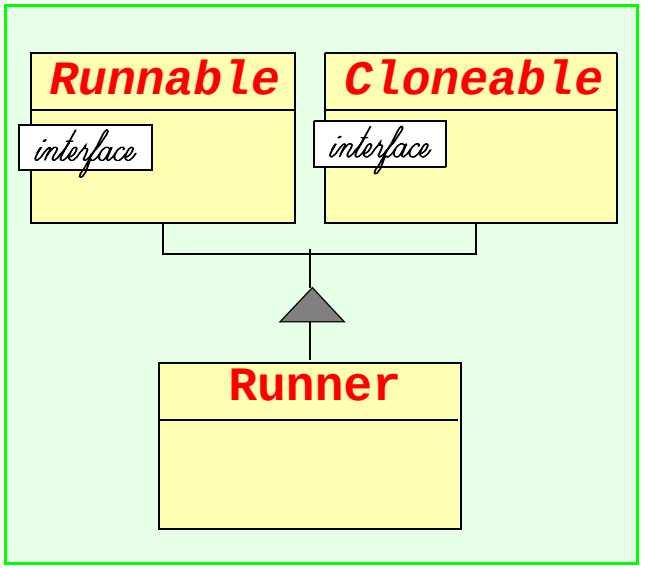
Java *Interface*: like a “Very Abstract Class”

interface is a *class* with all *methods* declared abstract and no *fields*

```
public interface Runnable{//abstract keyword omitted
     public void run();//abstract keyword omited
}
```

A class can *implement* one or more interfaces

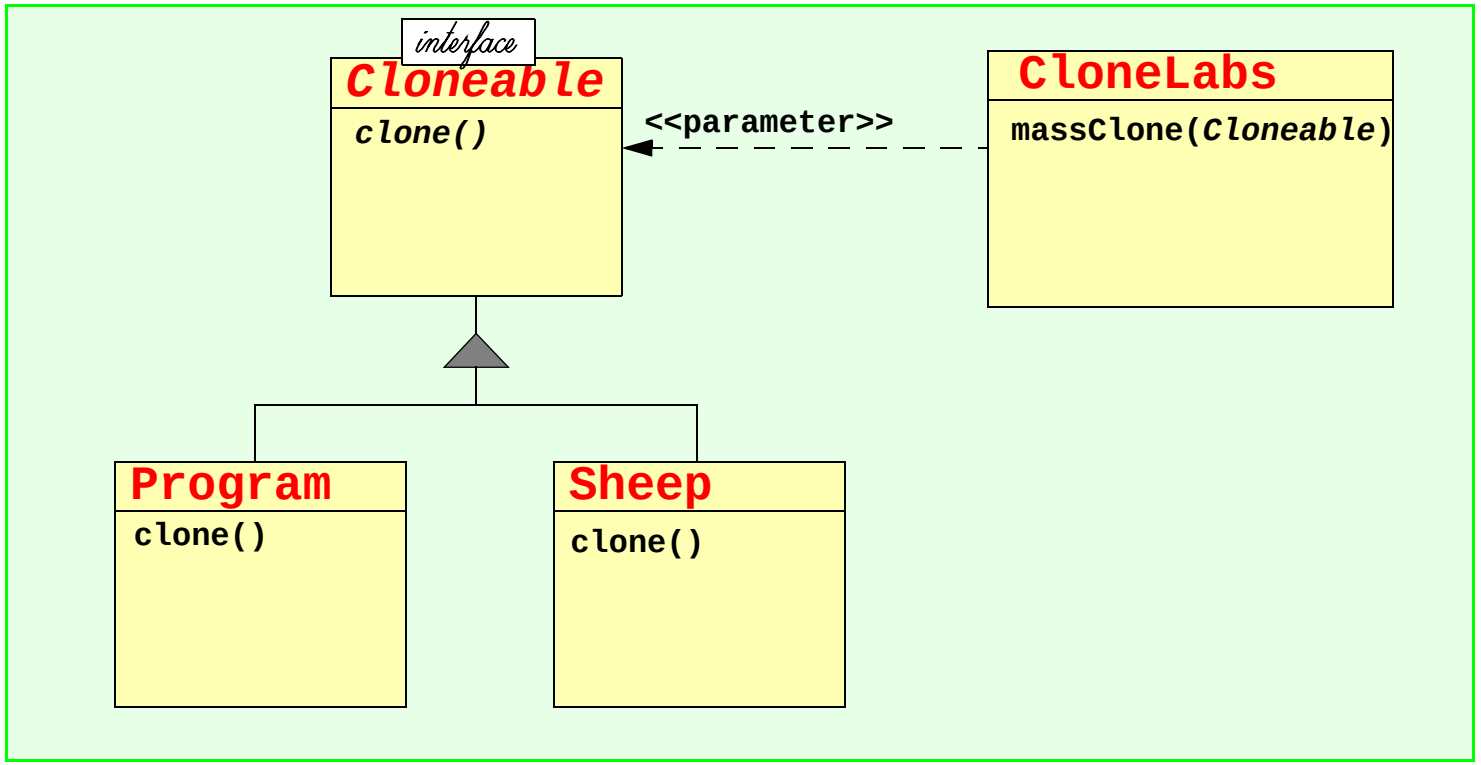
```
public interface Cloneable{
     public Object clone();
}

public class Runner implements Runnable,
                               Cloneable{
     public void run(){
          System.out.println(“Running”);
     }
     public Object clone(){
          return new Runner();
     }
}
```



Why interfaces? Example:

```
public interface Cloneable{ public Object clone(); }

public class Program implements Cloneable{
     public Object clone(){
          /* makes an identical deep copy and returns it */
     }
     /* ... and the rest of program class ... */
}

public class Sheep implements Cloneable{
     public Object clone(){
          /* makes an identical deep copy and returns it */
     }
     /* ... and the rest of sheep class ... */
}

public class CloneLabs{
     public Object[] massClone(Cloneable c){
          return new Object[]{c.clone(),c.clone(),c.clone()};
     }
}
```

Why interfaces? Example:



Inheritance, Abstracts and Interfaces

A class that extends an abstract class (or implements an interface) should provide a body for all the inherited *methods*, otherwise it becomes abstract also and must be declared abstract.

```
public abstract class AbstractRunner
                    implements Runner{}
```

is the same as

```
public abstract class AbstractRunner
                    implements Runner{
    public abstract void run();
}
```

An interface can only *extend* another interface, no other combinations allowed!

```
public interface CloneableRunner extends Cloneable,Runner{}
```

static members

static members (methods and fields) are **shared** between all instances of a class

```
public class Item{
    public static int count=0; //static field declaration
}

Item.count++; //static field access
```

static methods don't have associated instance

```
public static void setX(int value){
    this.x = value; //compile-time error
}
```

Static *field* can be seen like a "global" variable and static *method* like a "global" method:

field **`System.out`** or method **`System.println()`**

static members: example

```
public class Item{
     private static int lastId; //all items share it
     public int id;
     public Item(){ id=lastId++; }
}

new Item().id == 0 //true
new Item().id == 1 //true

public class A{
     private B b;
     public A(B other){ b=other; }
}
public class B{
     private A a;
     private B(){}
     public static B create( ){ B res = new B();
          res.a = new A(res); //accesses instance private field
          return res;}}

B.create(); //returns new couple A<->B
```

Life inside *methods*

Java method code syntax is very close to C/C++ :

```
public static void main(String[] args) {
     int sum = 0;//local var declaration
     for (int current = 1; current <= 10; current++) {//for loop
          sum += current;//plus and assign
     }
     System.out.println("Sum = " + sum);//prints: Sum = 55
}
```

Code is composed of statements terminated by "**;**"

```
System.out.println("Sum = " + sum); // a statement
```

Statements are grouped in blocks using "**{**" "**}**"

```
while(true){
     print(x); //I'm in a "while" block!
}
{ int x=0; } //and me I'm in a simple block ;-)
```

Life inside *methods* (II)

Local variables can be declared anywhere, they **must** be initialized, their scope is the enclosing block

```
int i=1;//i
{
    int j=i+1; //j==2
    int i=2; //compile-time error: i is already defined
}
int j=i;//another j==1
```

Method calls as usual:

```
out.write(“Salut”,22);
```

Operators and assignments very close to C++:

```
j=i++; a+=1; modulo=a%b; mask=f|g&h; flag=!g||h&&f;
```

Life inside *methods* (III)

Warning: all **logical** expressions are of type **boolean!!!** (and not int)

There is **no** automatic conversion boolean<->int

Some logical operators: < <= > >= ! == != && ||

```
(1<2)==true
((1<2)&&false)==false
(!true)==false
(1<2)==0 //compile-time error (type mismatch)
1+true //compile-time error (type mismatch)
```

Bitwise operators are of numeric types

Some bitwise operators: & | ~ >> << <<<

```
if((active & 0x20) != 0) ... //bitwise and, right
if(active & 0x20) ... //wrong!!!
```

Life inside *methods* (IV)

Control statements: **conditional** expressions are **boolean**! (and not int like in C)

```
if(flag){a();}else{b();} //if statement, flag is of type boolean!
if(flag){a();} //short form
for(int i=0;i<10;i++){ f(); } //condition is boolean
while(true){ f(); }//0+ times, condition is boolean
do{ f(); }while(true);//1+ times, condition is boolean

break;//exit from the enclosing block
continue;//jump to next iteration

return; //returns from a void method (procedure)
return x; //returns from a typed method (function)

switch(i){ //a switch
     case 1:f();break;
     case 2:g();break;
     default:e();
}
```

Life inside *methods* (V)

Use **this** keyword in order to distinguish between local variables and fields with same name

```
private String name;
public Person(String name){
     this.name = name; //initializes field name, cool style!
}
```

this keyword denotes current instance, so:

```
private int i;
public static foo(){
     this.i = 22; //compile-time error
}
```

Use **super** keyword in order to distinguish between original and redefined methods:

```
public void show(){print("Showing");super.show();}
```

Life inside *methods* (VI)

Class cast (type conversion) - must be used with caution

```
stack.push(“Salut”);
String s = (String)stack.pop();
```

Wrong cast throws a ClassCastException

```
String s = “AAA”;
f((NotString)s);//run-time ClassCastException!
```

Exceptions

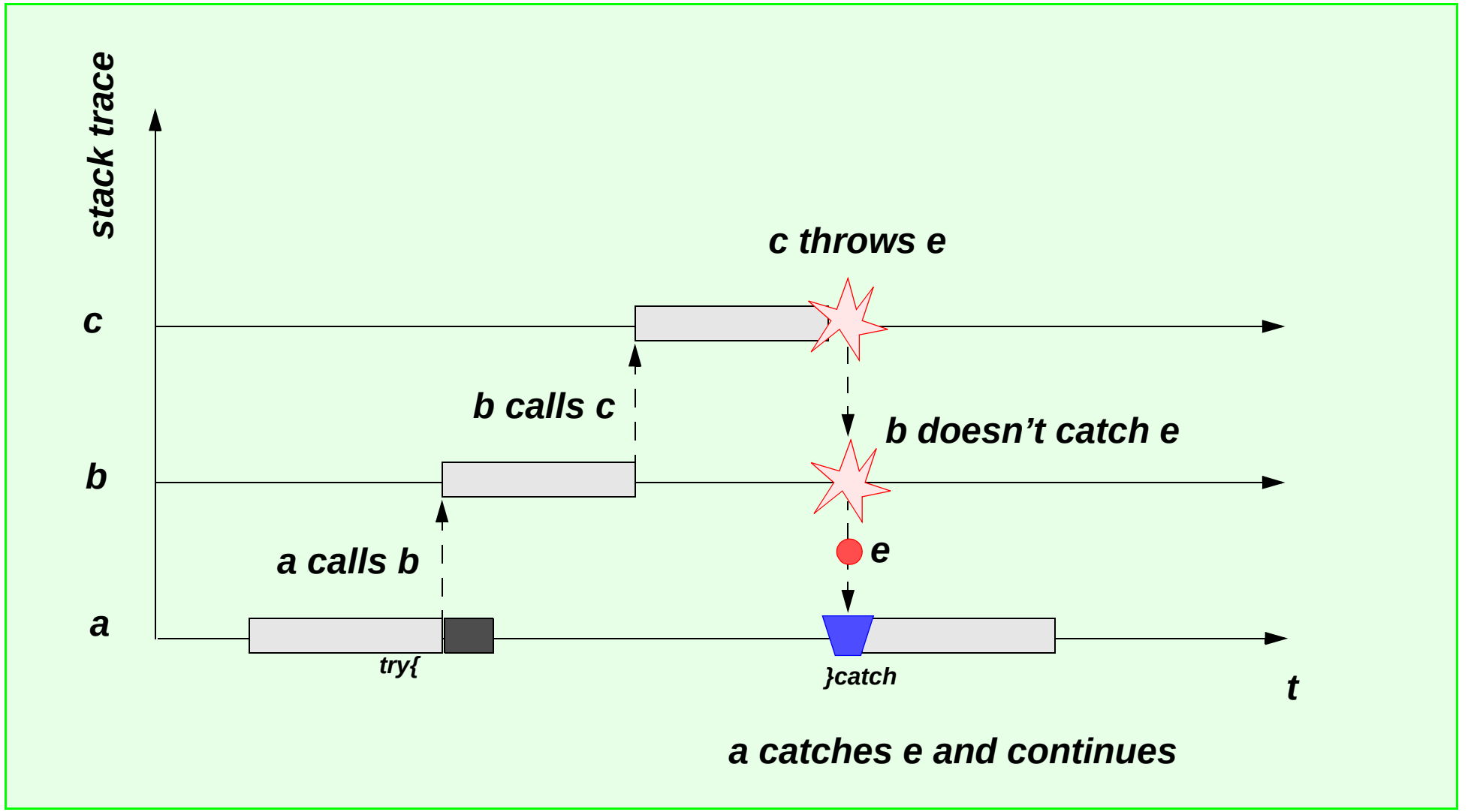
Indicate that a method didn't execute correctly

Good alternative to error codes in return values

Very convenient for multi threaded programs

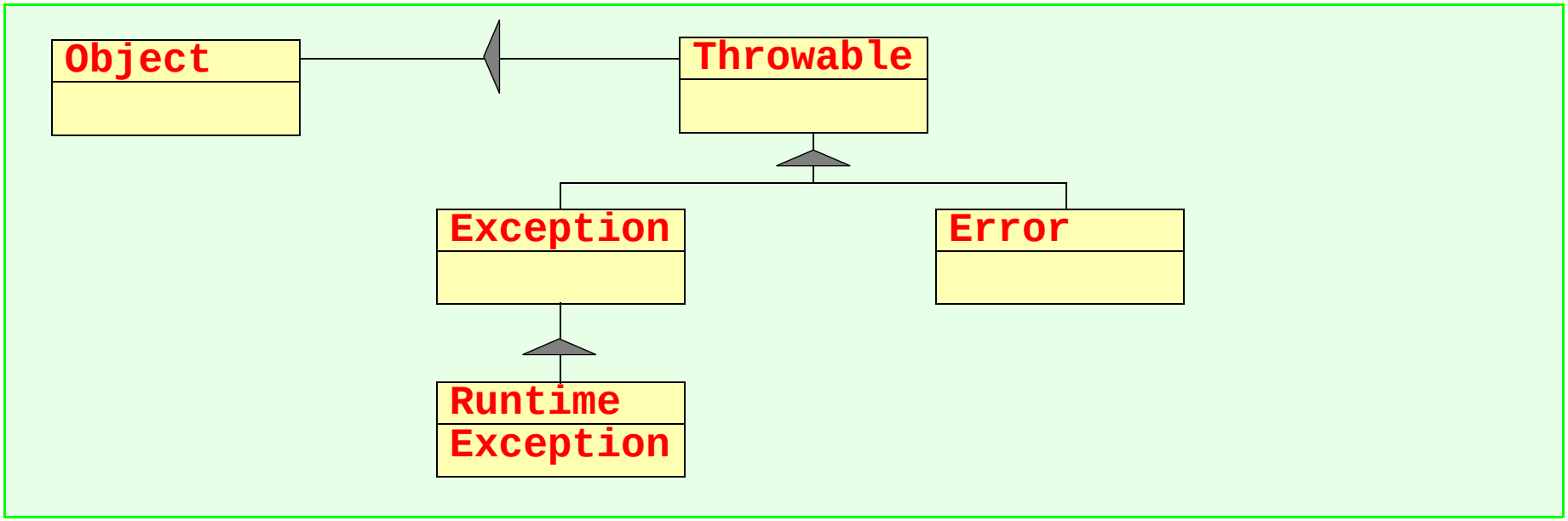
```
try{
    String s = in.readLine();//readLine can fail
    System.out.println( s );//case1:readLine succeed
}catch(IOException e){
    //case2:readLine failed
    System.err.println("Can not read");
}
```

Exceptions: how they works



Exceptions: class hierarchy

Java *exceptions* are *Objects*



All *Throwables* that a method throws **must** be declared, except *RuntimeExceptions* and *Errors*.

Usually *Runtimes* notify **bugs** and *Errors* notify **VM problems**

Examples of useful Exceptions.

1) **Runtime Exceptions**

- *NullPointerException*: attempt to invoke method on a null reference
- *IllegalArgumentException:* argument out of range
- *ArrayIndexOutOfBoundsException:* `a=new int[2];a[5]=1;`
- *ClassCastException:* `(String)new Object();`

2) **Exceptions** (extend *Exception* but not *RuntimeException*)

- *Exception:* any exception
- *IOException:* any IO problem

3) **Errors**

- *OutOfMemoryError:* VM runs out of memory

Exceptions: how to use

Exceptions (not R/T and not Errors) **must** be declared :

```
public void openDB() throws IOException{//declare it
     db=new FileInputStream("my.db"); //can fail, but don't care
}
```

... or catched :

```
public void openDB(){//don't declare it
     try{
          db=new FileInputStream("my.db"); //constructor can fail
     }catch(IOException e){//catch it
          //constructor fails
          System.err.println("No database found");
          System.exit(-1);//terminates execution
     }
}
```

Exceptions: how to use

You can throw exceptions yourself:

```
public Person(String name){
     if(name==null)throw new IllegalArgumentException(“Person must
have name“);
     this.name = name;
}
```

You can print exceptions:

```
     try{
          o.dangerous();
     }catch(MonsterCrashException mce){
          System.err.println(mce); //prints stack trace on stderr
     }
```

N*E*V*E*R do that in your code:

```
... }catch(Exception e){/* do nothing */} //very bad!!
```

Why it's so bad?

```
public class TerranSpacePort{
     public static void main(String[]argv){
          try{ new SpaceShip().launchTo("ZetaAquaelae");
          }catch(Exception e){}
     }
}

public class ZetaAquaelae extends DeepSpacePlanet{
     public void unload(){
          ...
          System.out.println("Unload started!");

unloadContainer(cont[decks[index+OFFSET]][row+opti-
mize(ROW_OFFSET)%priority]);//very optimized, but wrong unload
algorithm, throws
               //an ArrayIndexOutOfBoundsException!
          System.out.println("Unload finished");
     }
}
```

Output: `Unload started!` (...and nothing else) **You: "But my program blocks!?!"**

Object class

The common superclass of all Java classes

```
public class Object{
     public Class getClass() //the object's class

     public boolean equals(Object obj) //"semantic" equality
     public int hashCode() //"semantic" hash-code
     public String toString() // Object <-> String conversion

     /* Synchronization related methods */
     public void notify()
     public void notifyAll()
     public void wait(...)
     ...
}
```

Default `equals`: equality of references (== operator)

- **`equals`** and **`hashCode`** must be compatible:

`a.equals(b)` implies **`a.hashCode() == b.hashCode()`**

String Class

Java *Strings* are Objects

```
String s1 = “Hello”;
String s2 = new String(“Hello”);
s1 == s2 //false
s1.equals(s2) //true

s1.length() == 5 //true, don’t forget ()

“hello”.equals(word) //that’s ok!
“hello”.length()==5 //that’s ok too!
```

String can be concatenaded with any object or primitive values

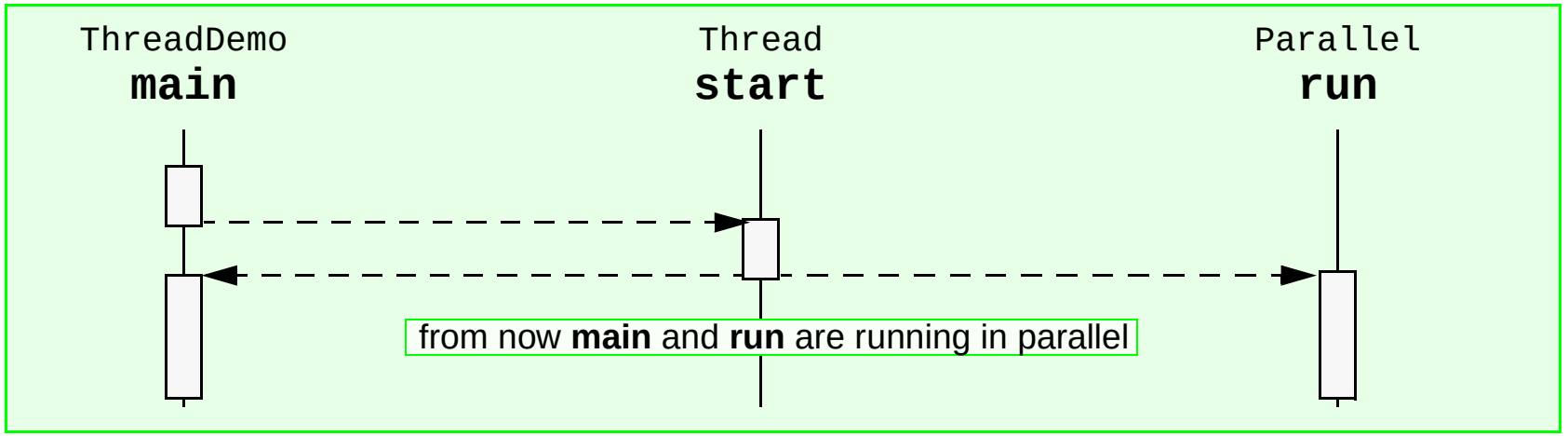
```
“there is “+5+” rooms in “+(new House());//concat
```

Java *Strings* are immutable! (for mutable version see StringBuffer)

Java *Threads* (virtual processors)

Use Thread class to create multiple parallel flows of control

```
public class Parallel implements Runnable{
     public void run(){
          System.out.println("Parallel flow started...");
          //... and continues execution ...
     }
}
public class ThreadDemo{
     public static void main(String[]argv){
          new Thread(new Parallel()).start();
          //... and continues ...
     }}
```



Objects and Threads

Each Java object has a ***lock*** (mutex) and a ***monitor*** (cond.variable)

Object's lock can be **free** or **owned** by at most one thread

a)Any thread can acquire a free lock (synchronized keyword)

b)If a thread tries to acquire a lock owned by another thread it's blocked until the lock becomes available

c)If a lock becomes free and there are many threads waiting for it, then only one thread will become the owner, choice is random

```
public synchronized void pop(){...}//synchronized method
public void pop(){
     ...
     synchronized(this){...}//synchronized block inside a method
     ...
}
```

Synchronization step by step

A synchronized method or block:

1. try to acquire the lock,

2. wait for it if lock is already owned,

3. becomes owner,

4. keep the lock when executing

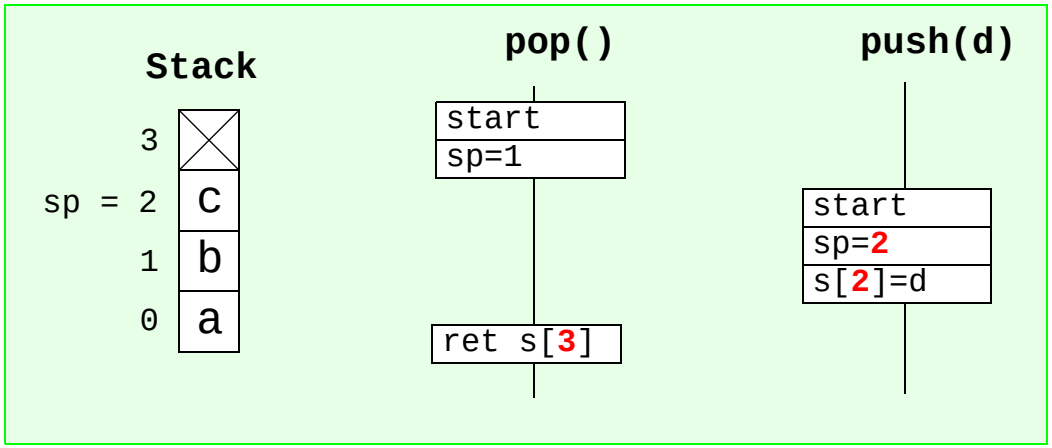
5. and finally releases the lock.

```
1,2,3:  synchronized(this){

   4:       //some code
            x = o.f(y);
            z++;

   5:   }
```

Synchronization

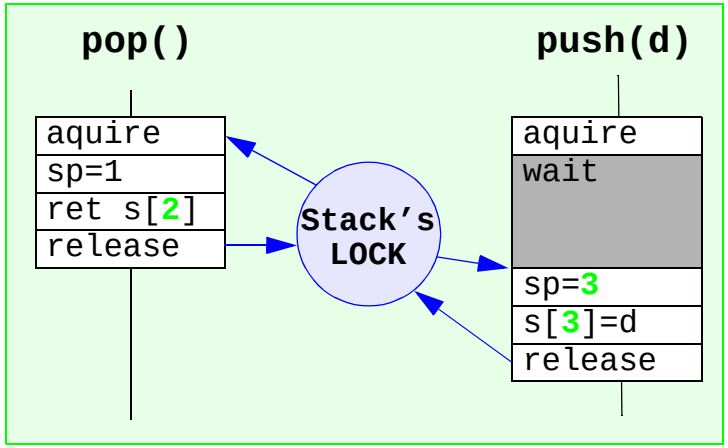
Synchronization: basic protection against concurrency problems

```
1    public void push(Object o){
2         sp=sp+1;
3         s[sp]=o;
     }

4    public Object pop(){
5         sp=sp-1;
6         return s[sp+1];
     }
```



"Synchronizing" methods remove concurrency

```
1    public synchronized void push(Object o){
2         sp=sp+1;
3         stack[sp]=o;
     }

4    public synchronized Object pop(){
5         sp=sp-1;
6         return stack[sp+1];
     }
```



Notification

One or more threads can **wait** (sleep) in object's monitor until another thread notifies them (wakes' em up)

To enter or exit a monitor a thread must acquire the object's lock.

Arrived in monitor thread go to sleep. **Lock is released**!

To exit the monitor, the notified thread (woken up) must first **reacquire the lock**

- **`notify()`** method wakes up one (random) thread
- **`notifyAll()`** method wakes up all threads

```
//executed by waiting thread
synchronized(o){
     o.wait();
}

//executed by notifying thread
o.notify()
...or...
o.notifyAll()
```

How to have fun with threads

Java threads are powerful but require discipline!

No discipline == **deadlocks**, **race conditions** and so on.

Acquiring multiple locks: always in same order

Choose between notify() or notifyAll() and be careful

```
/** Badly synchronized Buffer */
public class WrongBuffer extends Stack{
     private int capacity=10;
     public synchronized void put(Object o){
          if( size()==capacity )wait();
          push(o);
          notifyAll(); //all threads blocked in get
     }
     public synchronized Object get(Object o){
          if(isEmpty())wait();
          Object ret = pop();
          notifyAll();
          return ret;
     }}
```

How to have fun with threads(II)

Doing well is easy:

```
/** Well synchronized Buffer */
public class GoodBuffer extends Stack{
     private int capacity=10;
     public synchronized void put(Object o){
          while( size()==capacity )wait(); //buffer full, wait for a get
          push(o);
          notifyAll(); //means all threads blocked in get
     }

     public synchronized Object get(Object o){
          while( isEmpty() ) wait(); //buffer empty, wait for a put
          Object ret = pop();
          notifyAll(); //means all threads blocked in put
          return ret;
     }
}
```

How to have fun with threads(III)

Another way:

```
/** Well synchronized Buffer */
public class GoodBuffer extends Stack{
     private int capacity=10;
     public synchronized void put(Object o){
          if( size()==capacity )wait(); //buffer full, wait for a get
          push(o);
          notify(); //one of threads blocked in get
     }

     public synchronized Object get(Object o){
          if( isEmpty() ) wait(); //buffer empty, wait for a put
          Object ret = pop();
          notify(); //one of threads blocked in put
          return ret;
     }
}
```

How to use classes from another package

Use full class name (prefixed with package name)

```
java.awt.Frame f= new java.awt.Frame("Hello"); //long names ;(
```

... or put an import statement in the beginning of the class

```
package myPackage;

import java.awt.*;

public class Main{
    public static void main(String argv[]){
        Frame f= new Frame("Hello"); //long names ;(
    }
}
```

Compiler will automaticaly resolve ***Frame*** as ***java.awt.Frame***

- `java.lang` package is explicitly imported

Essential Java Libraries

java.lang: the core of core libraries

java.util: *collections* and some other useful classes

java.io: Java IO Library, stream-based

java.net: Java Network Library

java.awt: Abstract Windows Toolkit: basic GUI library

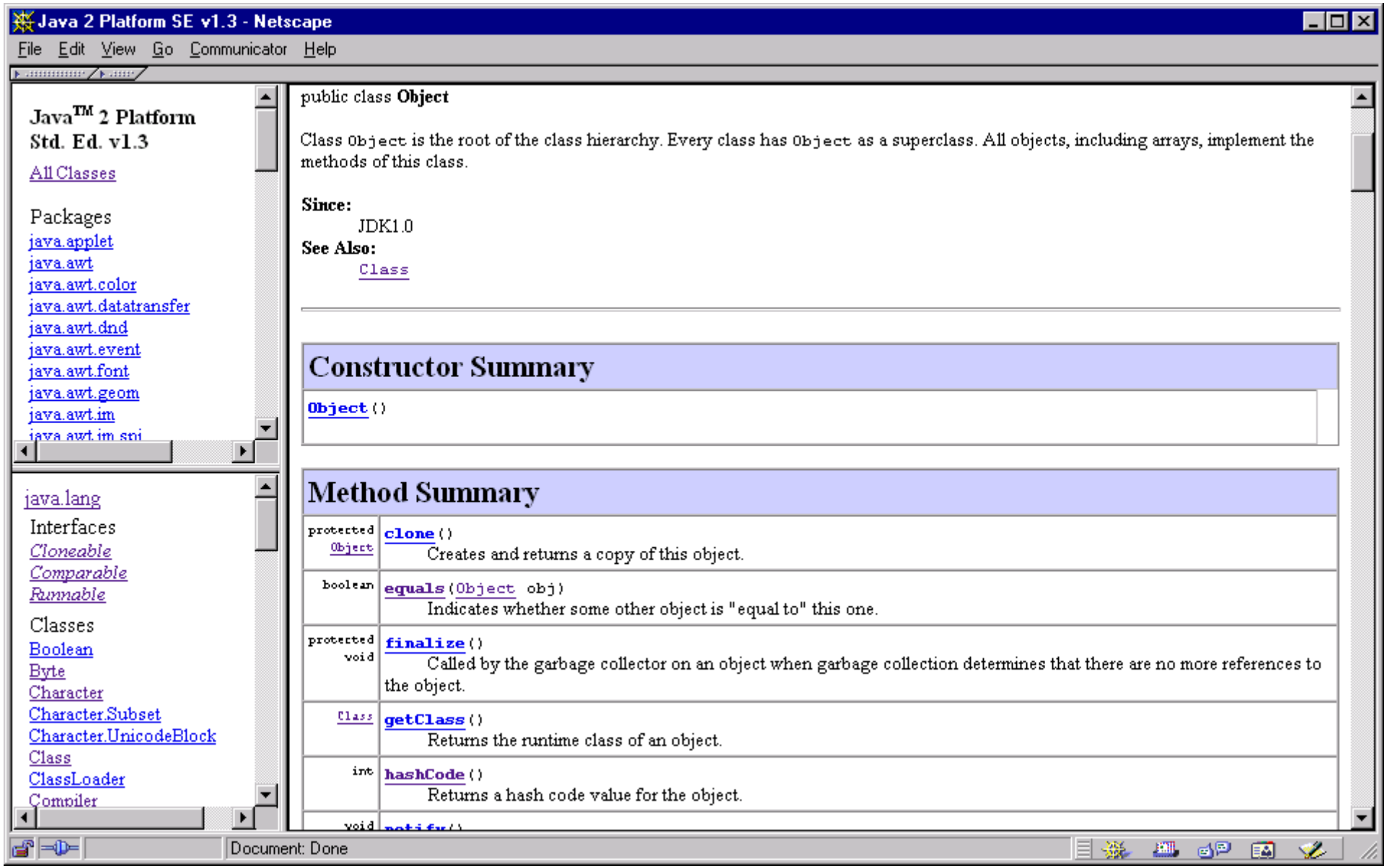
javax.swing: lightweight and powerful GUI library based on AWT

JavaDoc(umentation)

Thanks to structured comments class documentation can be automatically generated:

```
/**
* House protection
*/
public class Door{

     /**
     * Lock's state
     */
     private boolean open=false;

     /**
     * Door status
     * @returns true is the door is open
     */
     public boolean isOpen(){
          return open;
     }
}
```

What HTML JavaDocs look like?



java.lang.System

Essential OS functions are in this class here

`System.in`, `System.out` and `System.err` : Java program's `stdin`, `stdout` and `stderr`. Can print any java value (object or primitive)

```
System.out.println("Hello world!");
```

`System.exit(int status)` : exit program and return a status

```
System.exit(0); //normal termination
```

`System.arraycopy()`: fast array copy

```
System.arraycopy(a1,0,a2,10,5); //copy a1[0-4] to a2[10-14]
```

`System.getProperty(String key)`: retrieves a property (predefined or specified by *-Dkey=value* on command line)

```
System.getProperty("user.dir"); //program's working directory
```

java.lang.Thread

`Thread(Runnable r)`: creates a new thread with specified Runnable

```
Thread t1 = new Thread(new MyTask()); //ready to run,but not started yet
```

`start()`: starts the thread

```
t1.start(); //starts MyTask starts in parallel
```

`setPriority(int prio)`: sets the thread's priority, used by scheduler

```
t1.setPriority(Thread.MIN_PRIORITY); //minimal priority
```

`join(),Thread.sleep(long),Thread.yield()`: as usual

```
t1.join(); //current thread will wait for t1 to die
Thread.sleep(1000); //current thread will sleep at least 1 second
```

`stop(),suspend(),resume()`: deadlock-prone - **don't use!**

java.lang.Integer and other *wrappers*

Are used to “wrap” a primitive value into Object

```
Integer ii = new Integer(10);
int i=ii.intValue(); //i==10
new Character(‘b’)
Boolean.TRUE
```

Contain some useful method for manipulate primitive values

```
Character.isLowerCase(‘A’) //false
```

Contain parser methods

```
Integer.parseInt(“A0”,16) //returns 160
```

Java Characters

Java characters (**`char`**) are coded in **Unicode** (two bytes per char) - all alphabets accepted

```
“La sûreté et la vivacité”
```

Java identifiers (variable, method etc.) are also in Unicode

```
public int sûreté = -12;
```

Collections (in java.util)

A ***Collection*** interface represents a group of objects, known as its elements, duplicated or not, ordered or unordered.

Implemented by several classes: **ArrayList**, **HashMap** etc.

`add(Object o)`: add an object to collection

`boolean contains(Object o)`: tests if o is in collection

`int size()`: collection's size

`remove(Object o)`: remove element from collection

`clear()`: remove all elements

`Iterator iterator()`: returns an Iterator

`Object[] toArray()`: collection contents in array

java.util.Iterator

An iterator over a collection. Only three methods:

- **`boolean hasNext()`**: true if the iteration has more elements
- **`Object next()`**: return next element of the iteration
- **`remove()`**: remove last returned element from underlying collection

```
import java.util.*;
...
Collection c=...

Iterator i = c.iterator();

while(c.hasNext()){
     System.out.println("Element:" + c.next());
}
```

java.util.HashMap

A collection of objects (values) indexed by objects (keys)

- **`put(Object key,Object value)`**: stores value indexed by key
- **`Object get(Object key)`**: returns value mapped by key
- **`int size()`**: current map's size
- **`Iterator iterator()`**: iterates over mappings
- **`Iterator iterator()`**: iterates over mappings
- **`Collection keys(),Collection values()`**: keys, values

```
import java.util.*;
...
HashMap dict= new HashMap();
dict.put("One", "Un"); //maps "One" to "Un"
dict.put("Cat", "Chat");//maps "One" to "Un"

String fr_cat = (String)dict.get("Cat");//fr_cat constains "Chat",cast!
dict.size() == 2//true
```

java.util.ArrayList

Resizable array of Objects (int-indexed, zero based).

- **`add(Object o)`**: append o at the end of the list
- **`Object get(int i)`**: return i-th element. Cast is frequently used.
- **`int size()`**: current list's size
- **`remove(int i), remove(Object o)`**: remove element
- **`set(int i,Object o)`**: puts o at index i (overwrite)
- **`Iterator iterator()`**: iterates over elements

```
import java.util.*;
...
ArrayList list = new ArrayList();
list.add("Hello world"); //append at te end
list.add(new Character('!')); //wrappers are useful

list.size()==2 //true
String hello = (String)list.get(0);//first element, cast needed
```

java.util.Date and java.util.Calendar

Time manipulation classes.

Date represent the specific instant in time with millisecond precision.

```
Date t1 = new Date();
f();
Date t2 = new Date();
System.out.println("f takes "+(t2.getTime()-t1.getTime())+" ms.");
```

Example output: “f takes 17231 ms.”

Calendar interpret a Date according to the rules of a specific calendar system.