Introduction to Java

Università di Modena e Reggio Emilia
Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

The Java Environment

Java Timeline

- 1991: SUN develops a programming language for cable TV set-top boxes
- 1994: Java-based web browser (HotJava), the idea of “applet” comes out
- 1996: first version of Java (1.0)
- 1996: Netscape supports Java. Popularity grows
- 1998: Java 2 platform (1.2 ver) released (libraries)
- 2005: Java 5 (language enhancements)

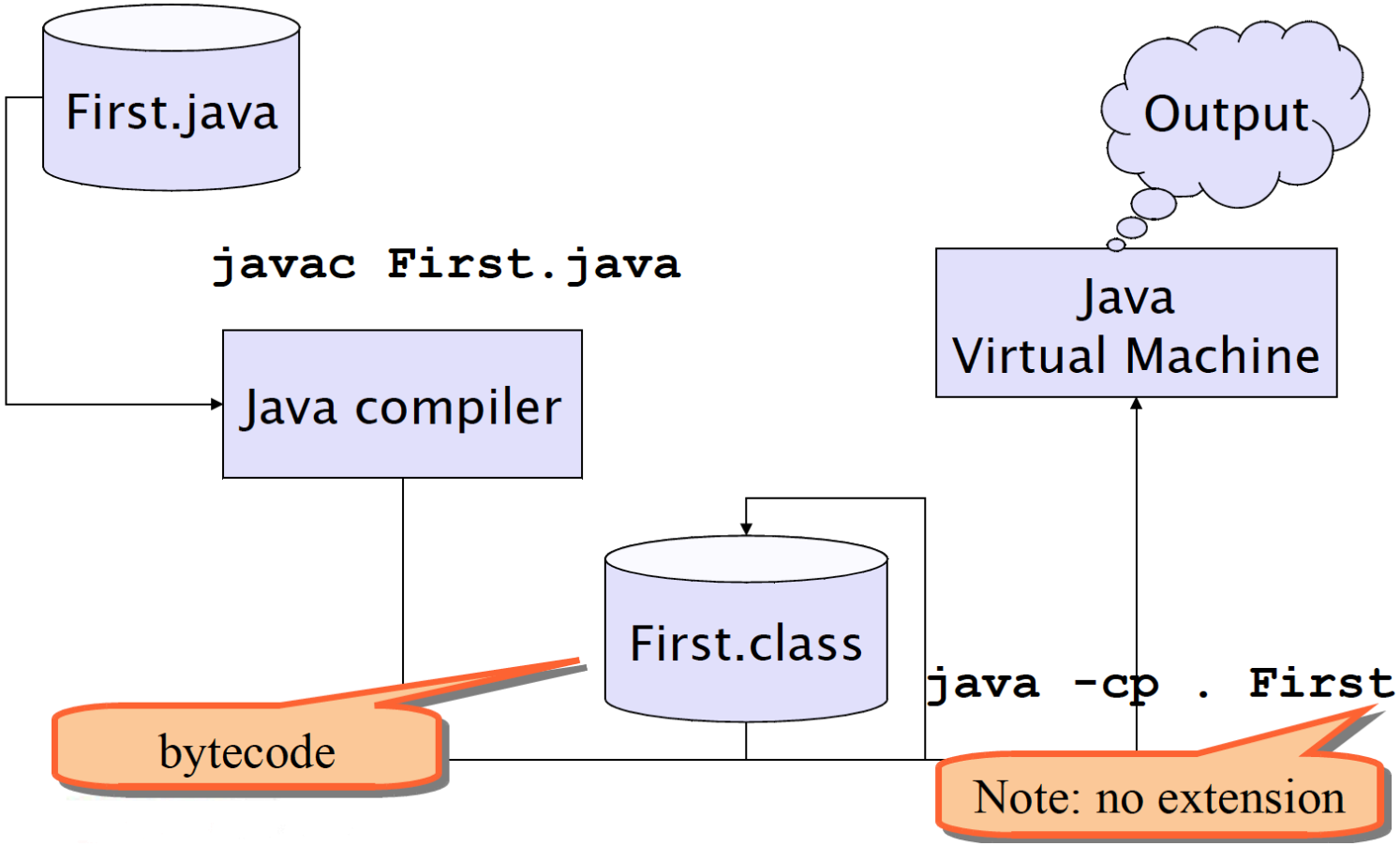
Java Features

- Platform independence (portability)
  - Write once, run everywhere
  - Translated to intermediate language (bytecode)
  - Interpreted (with optimizations, e.g. JIT)
- Run time loading and linking
- Automatic garbage collection
- Robust language, i.e. less error prone
  - Strong type model and no pointers
  - Exceptions as a pervasive mechanism to
- Lots of standard utilities included
- Shares many syntax elements w/ C++
  - Learning curve is less steep for C/C++ Programmers
- Pure OO language

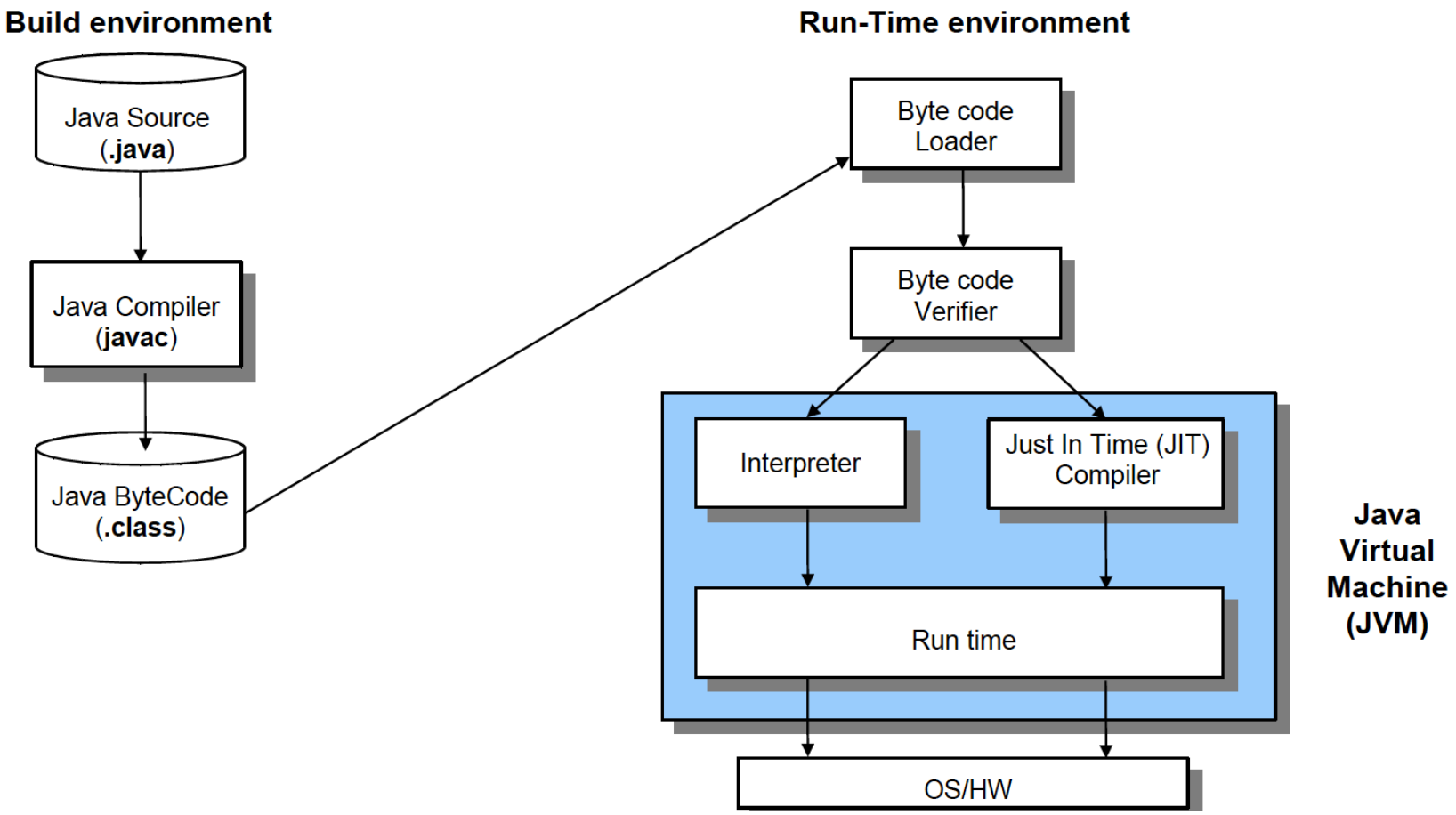
Java Programs

- Application
  - It's a common program, similarly to C
  - executable programs
  - Runs through the Java interpreter (java) of the installed Java Virtual Machine
- Applet (client browser)
  - Java code dynamically downloaded
  - Execution is limited by "sandbox"
- Servlet (web server)
  - In J2EE (Java 2 Enterprise Edition)
- Midlet (mobile devices, e.g. smartphone and PDA)
  - In J2ME (Java 2 Micro Edition)

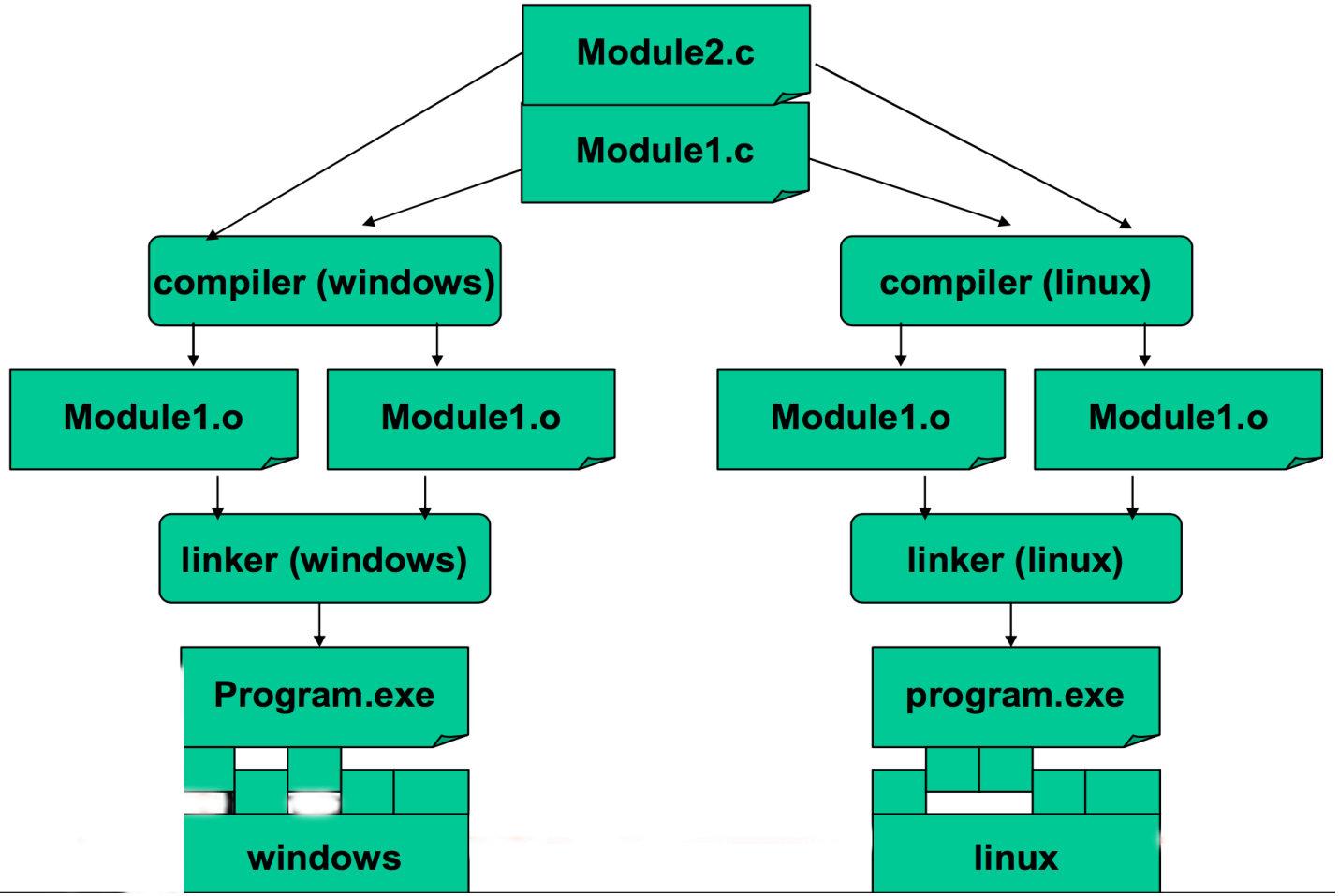
Building and running



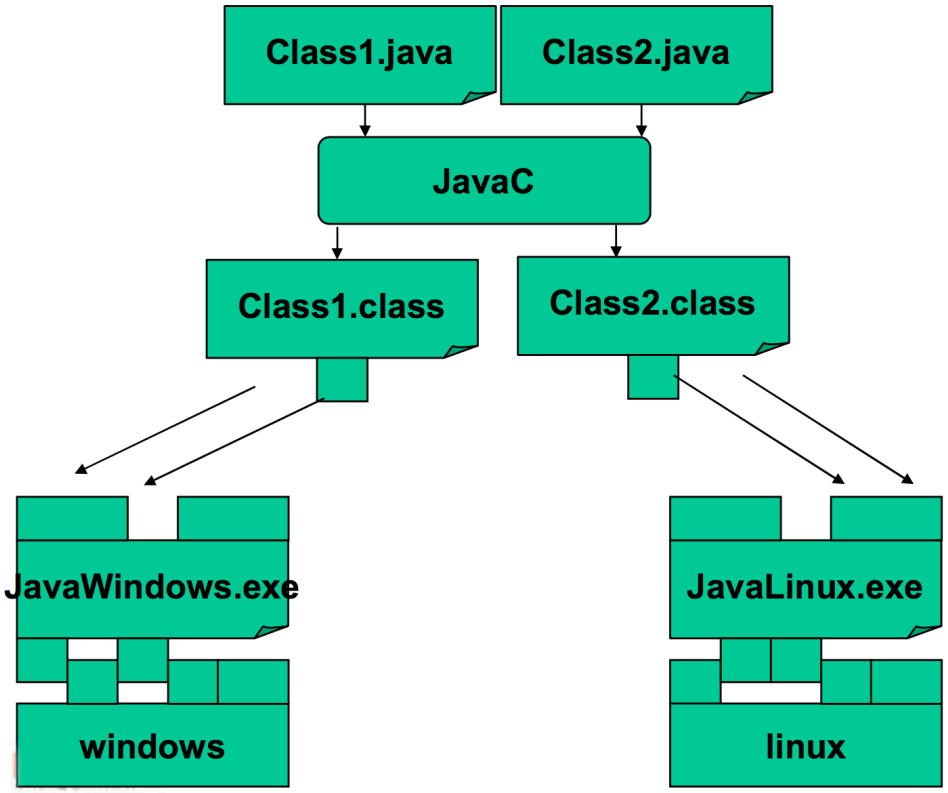
Building and running



C (Compiled)


Module2.c
Module1.c
compiler (windows)
compiler (linux)
Module1.o
Module1.o
Module1.o
Module1.o
linker (windows)
linker (linux)
Program.exe
program.exe
windows
linux

Java (Interpreted)



Coding Conventions

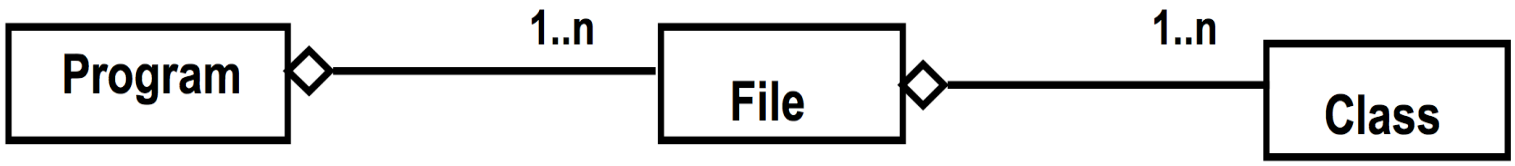
```
class ClassName {
    const double PI = 3.14;
    private int attributeName;
    public void methodName {
        int var;
        if ( var==0 ) {
            /* this is comment*/
        }
    }
}
```

public static void main(String[] args)

- In Java there are no functions, but only methods within classes
- The execution of a Java program starts from a special method:
- ***public static void main(String[] args)***
- Note
  - return type is **void**
  - **args[0]** is the first argument on the command line (after the program name)

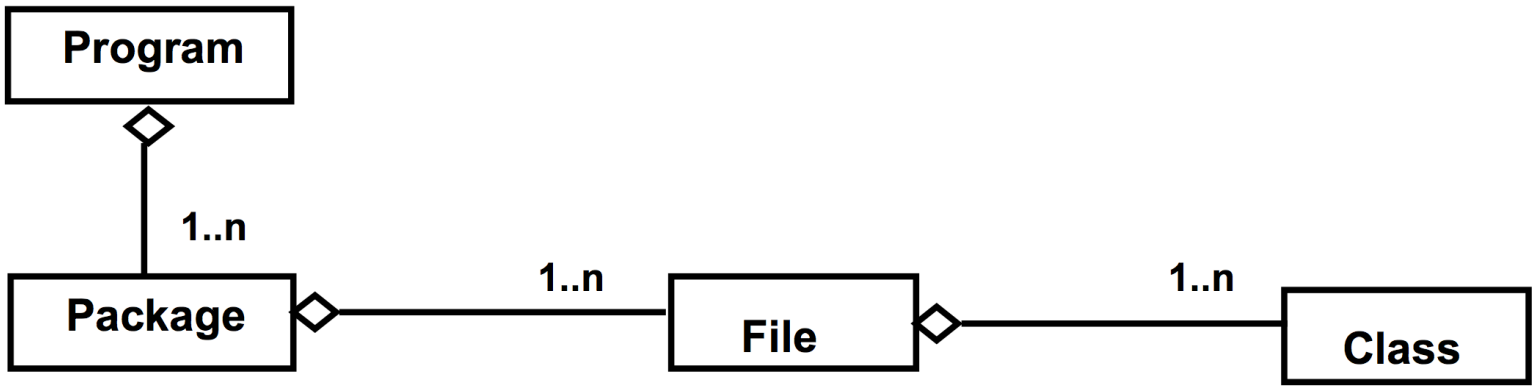
Program, files and classes

- A program is made of many classes
- A class is in one file
- A file usually contains one class
  - Can also contain more than one, but only one public
- Name of file must be == name of public class



Packages

- Packages add one more level
- Package is implemented via a directory
  - name of directory == name of package



Basic concepts

Comments

- C-style comments (multi-lines)

```
/* this comment is so long
that it needs two lines */
```

- Comments on a single line

```
// comment on one line
```

Code blocks and Scope

- Java code blocks are the same as in C language
- Each block is enclosed by braces { } and starts a new scope for the variables
- Variables can be declared both at the beginning and in the middle of block code

```
for (int i=0; i<10; i++){
    int x = 12;
    ...
    int y;
    ...
}
```

Control statements

- Same as C
  - if-else,
  - switch,
  - while,
  - do-while,
  - for,
  - break,
  - continue

Boolean

- Java has an explicit type (boolean) to represent logic values (true, false)
- Conditional constructs evaluate boolean conditions

Passing Parameters

- Parameters are always passed by value
- ...they can be primitive types or object references
- Note well: only the object reference is copied not the value of the object

Constants

- The final modifier

```
final float PI = 3.14;
PI = 16.0;          // ERROR, no changes
final int SIZE;     // ERROR, init missing
```

- Use uppercases (coding conventions)

Classes and primitive types

- Class

```
class Exam {}
```

descriptor

- type primitive

```
int, char,
  float
```

- Variable of type reference

```
Exam e;

e = new Exam();
```

instance

- Variable of type primitive

```
int i;
```

Class

- Defined by developer (e.g., Exam) or by the Java environment (e.g., String)

```
Exam e;
```

- Allocates memory space for the reference.

```
e = new Exam();
```

- Allocates and initialize the object value (constructor)

Primitive Types

Primitive types

- Unique dimension and encoding
- Platform-independent representation

type	Dimension		Encoding
boolean	1	bit	–
char	16	bits	Unicode
byte	8	bits	Signed integer 2C
short	16	bits	Signed integer 2C
int	32	bits	Signed integer 2C
long	64	bits	Signed integer 2C
float	32	bits	IEEE 754 sp
double	64	bits	IEEE 754 dp
void	–	–	

Constants

- Constants of type int, float, char, strings follow C syntax
  - 123 256789L 0xff34 123.75 0.12375e+3
  - “a” “%” “\n” “prova” “prova\n”
- Boolean constants (do not exist in C) are
  - true, false

Operators (integer and floating-point)

- Operators follow C syntax:
  - arithmetical + - * / %
  - relational == != > < >= <=
  - bitwise (int) & ! >> << ~
  - Assignment = += -= *= /= %= &= |= ^=
  - Increment ++ --
- Chars are considered like integers (e.g. switch)

Logical operators

- Logical operators follows C syntax:

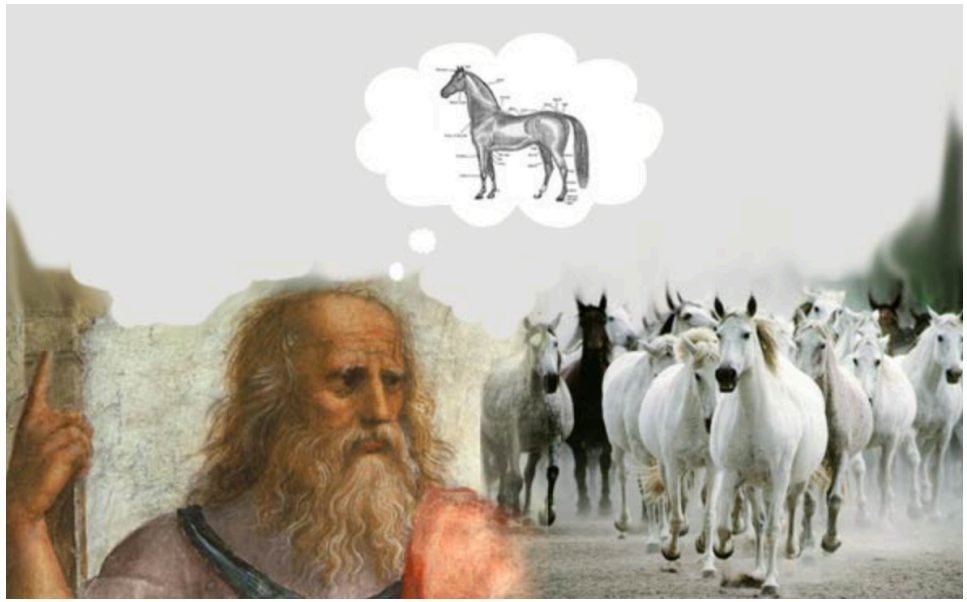
  && || ! ^

- Note well: Logical operators work ONLY on booleans. Type int is NOT considered a boolean value like in C

Classes

Class

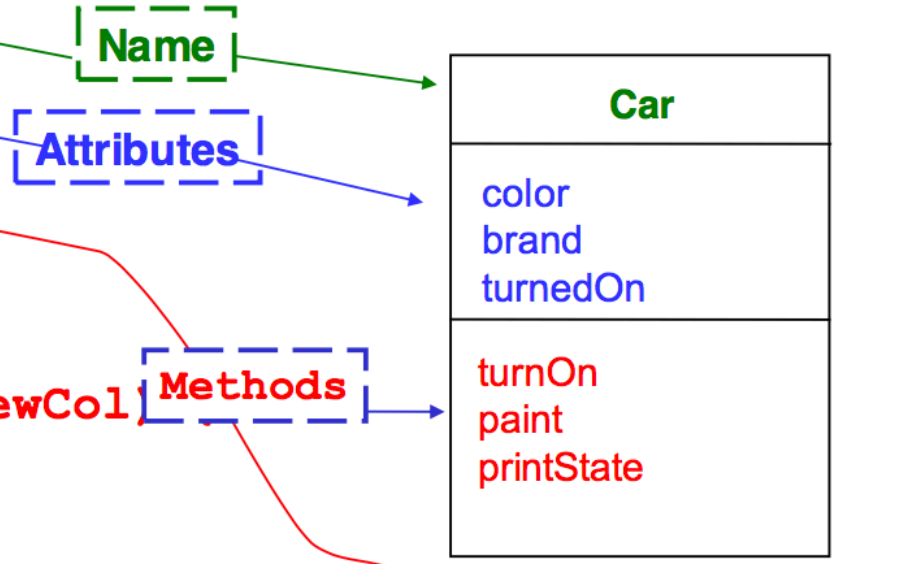
- Object descriptor *(Platonic idea)*



- It consists of attributes and methods

Definition

```
class Car {
  String color;
  String brand;
  boolean turnedOn;
  void turnOn() {
    turnedOn = true;
  }
  void paint (String newCol) {
    color = newCol;
  }
  void printState () {
    System.out.println("Car " + brand + " " + color);
    System.out.println("the engine is"
      +(turnedOn:"on":"off"));
  }
}
```



Methods

- Methods are the messages that an object can accept
  - **turnOn**
  - **paint**
  - **printState**
- Method may have parameters
  - **paint(“Red”)**

Overloading

- In a Class there may be different methods with the same name
- But they have a different signature
- A signature is made by:
  - Method name
  - Ordered list of parameters types
- The method whose parameters types list matches, is then executed

```
class Car {
  String color;
  void paint(){
    color = "white";
  }
  void paint(int i){}
  void paint(String
             newCol){
    color = newCol;
  }
}
```

Overloading

```
public class Foo{
  public void doIt(int x, long c){
    System.out.println("a");
  }
  public void doIt(long x, int c){
    System.out.println("b");
  }
  public static void main(String args[]){
    Foo f = new Foo();
    f.doIt(       5 ,(long)7 ); // "a"
    f.doIt( (long)5 ,       7 ); // "b"
  }
}
```

Objects

- An object is identified by:
  - Its class, which defines its structure
  - (attributes and methods)
  - Its state (attributes values)
- An internal unique identifier
- Zero, one or more reference can point to the same object

Objects and references

```
Car a1, a2; // a1 and a2 are uninitialized
a1 = new Car();
a1.paint("yellow");
// a1 is a reference pointing to "yellow"

a2 = a1; //now two references point to "yellow"
a2 = null; // a reference points to "yellow"
a1 = null;// no more references point to "yellow"

// Object exists but it is no more reachable
// then it will be freed by
// the garbage collector
```

Objects creation

- Creation of an object is made with the keyword new
- It returns a reference to the piece of memory containing the created object
- **`Motorcycle m = new Motorcycle();`**

The keyword new

- Creates a new instance of the specific Class, and allocates the necessary memory in the heap
- Calls the constructor method of the object (a method without return type and with the same name of the Class)
- Returns a reference to the new object created
- Constructor can have parameters
  - `String s = new String("ABC");`

Constructor

- Constructor method contains operations (initialization of attributes etc.) we want to execute on each object as soon as it is created
- Attributes are always initialized
  - Attributes are initialized with default values
- If a Constructor is not declared, a default one (with no parameters) is defined
- Overloading of constructors is often used

Destruction of objects

- It is no longer a programmer concern
- See section on Memory management
- Before the object is destroyed - if exists - method finalize is invoked:

```
public void finalize()
```

Access to methods

- A method is invoked using dotted notation

objectReference.Method(parameters)

- Example:

```
Car a = new Car();
a.turnOn();
a.paint("Blue");
```

Access to attributes

- Dotted notation
  - ***objectReference.attribute***
  - Reference is used like a normal variable

```
Car a = new Car();
a.color = “Blue”; //what‟s wrong here?
boolean x = a.turnedOn;
```

Local attributes and methods

- Methods accessing attributes or methods of the same object do not need using object reference (this implied)

this

- It can be useful in methods to distinguish object attributes from local variables
  - this represents a reference to the current object

```
class Car{
   String color;
   ...
   void paint (String color) {
      this.color = color;
   }
}
```

Combining dotted notations

- Dotted notations can be combined
  - **System.out.println("Hello world!");**
- **System** is a Class in package java.lang
- **Out** is a (static) attribute of System referencing an object of type **PrintStream** (representing the standard output)
- **Println()** is a method of **PrintStream** which prints a text line on the screen

Operations on references

- Only the relational operators == and != are defined
  - Note well: the equality condition is evaluated on the values of the references and NOT on the values of the objects !
  - The relational operators tell you whether the references points to the same object in memory
- Dotted notation is applicable to object references
- There is NO pointer arithmetic

Strings

String

- No primitive type to represent string
- String literal is a quoted text C
  - **char s[] = “literal”**
  - Equivalence between string and char arrays
- Java
  - **char[] != String**
  - java.lang.String (see Java API)

Operator +

- It is used to concatenate 2 strings
  - **“This string” + ” is made by two strings”**
- Works also with other types (automatically converted to string)
  - **System.out.println(”pi = ” + 3.14);**
    System.out.println(”x = ” + x);

StringBuffer

- Useful for concatenating Strings.
- A thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls. *(insert(), append(), delete(), reverse(), toString(),…)*

Character

- Utility methods on the kind of char
  - **isLetter(), isDigit(), isSpaceChar()**
- Utility methods for conversions
  - **toUpper(), toLower()**

Scope and encapsulation

Example

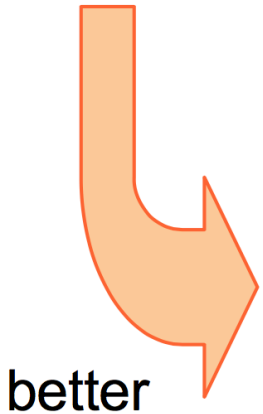
- Laundry machine, design1
  - commands:
    - time, temperature, amount of soap
  - Different values depending if you wash cotton or wool, ...
- Laundry machine, design2
  - commands:
    - key C for cotton, W for wool, Key D for knitted robes

Example

- Laundry machine, design3
  - commands:
    - Wash!
  - insert clothes, and the washing machine automatically select the correct program
- There are different solutions with different level of granularity / abstraction

Information hiding

```
class Car {
  public String color;
}
Car a = new Car();
a.color = "white";  // ok
```



better

```
class Car {
  private String color;
  public void paint(String color)
    {this.color = color;}
}
Car a = new Car();
a.color = "white";  // error
a.paint("green");   // ok
```

Access

	Method in the same class	Method of another class
Private (attribute/ method)	yes	no
Public	yes	yes

Getters and setters

- Methods used to read/write a private attribute
- Allow to better control in a single point each write access to a private field

```
public String getColor() {
    return color;
}
public void setColor(String newColor) {
    color = newColor;
}
```

Example without getter/setter

```
public class Student {
    public String first;
    public String last;
    public int id;
    public Student(...){...}
}
public class Exam {
    public int grade;
    public Student student;
    public Exam(...){...}
}
```

Example without getter/setter

```
class StudentExample {
        public static void main(String[] args) {
                // defines a student and her exams
                // lists all student‟s exams
                Student s=new Student(“Alice",“Green",1234);
                Exam e = new Exam(30);
                e.student = s;
                // print vote
                System.out.println(e.grade);
                // print student
                System.out.println(e.student.last);
        }
}
```

Example with getter/setter

```
class StudentExample {
   public static void main(String[] args) {
      Student s = new Student(“Alice”, “Green”,
1234);
      Exam e = new Exam(30);
      e.setStudent(s);
      // prints its values and asks students to
      // print their data
      e.print();
   }
}
```

Example with getter/setter

```
public class Student {
    private String first;
    private String last;
    private int id;
    public String toString() {
        return first + " " + last +""+ id;
    }
}
```

Example with getter/setter

```
public class Exam {
    private int grade;
    private Student student;
    public void print() {
        System.out.println("Student " +
            student.toString() + "got " + grade);
    }
    public void setStudent(Student s) {
        this.student = s;
    }
}
```

Array

Array

- An array is an ordered sequence of variables of the same type which are accessed through an index
- Can contain both primitive types or object references (but no object values)
- Array dimension can be defined at run-time, during object creation (cannot change afterwards)

Array declaration

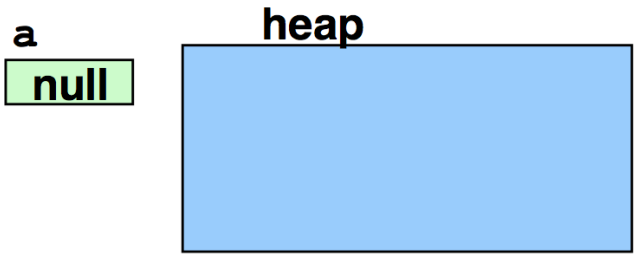
- An array reference can be declared with one of these equivalent syntaxes
  - int[] v, int v[]
- In Java an array is an Object and it is stored in the heap
- Array declaration allocates memory space for a reference, whose default value is null

Array creation

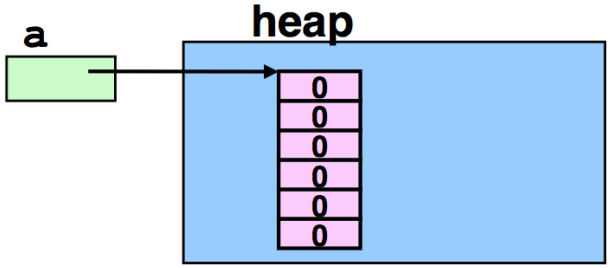
- Using the new operator
  - int[] v = new int[256];
- Using static initialization, filling the array with values
  - int[] v = {2,3,5,7,11,13}

Example – Primitive types

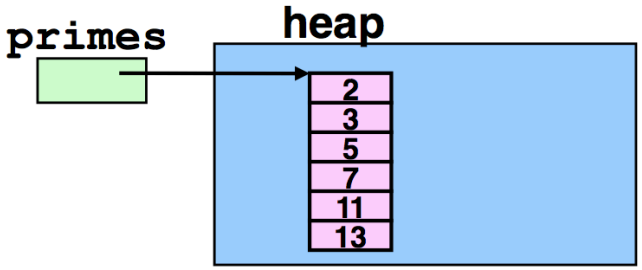
```
int[] a;
```




```
a = new int[6];
```

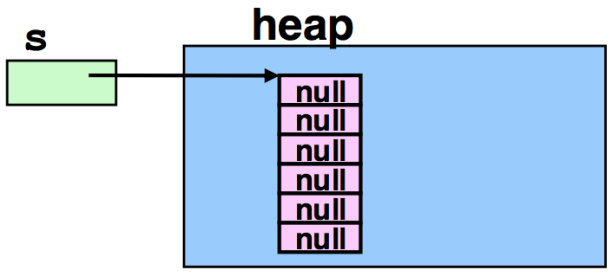



```
int[] primes =
  {2,3,5,7,11,13};
```

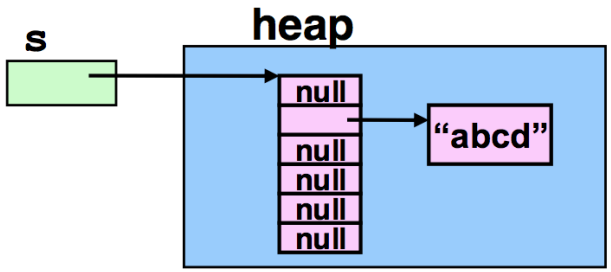


Example – Object reference

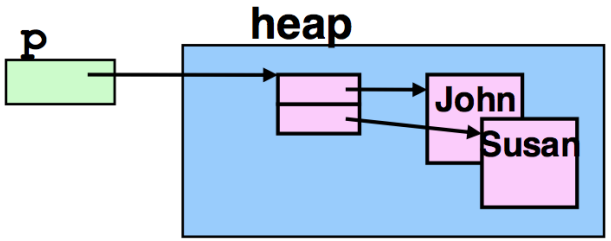
```
String[] s = new
  String[6];
```




```
s[1] = new
  String("abcd");
```




```
Person[] p =
{new Person("John") ,
 new Person("Susan")};
```



Operations on arrays

- Elements are selected with brackets [ ] (C-like)
  - But Java makes bounds checking
- Array length (number of elements) is given by attribute length

for (int i=0; i < a.length; i++) a[i] = i;

Operations on arrays

- An array reference is not a pointer to the first element of the array
- It is a pointer to the array object
- Arithmetic on pointers does not exist in Java

Operations on arrays

- New loop construct:
  for(*Type var* : *set_expression*)
- Notation very compact set_expression can be either
  - an array
  - a class implementing **Iterable**
- The compiler can generate automatically loop with correct indexes
  - less error prone

For each

```
for(String arg: args){
   //...
}
```

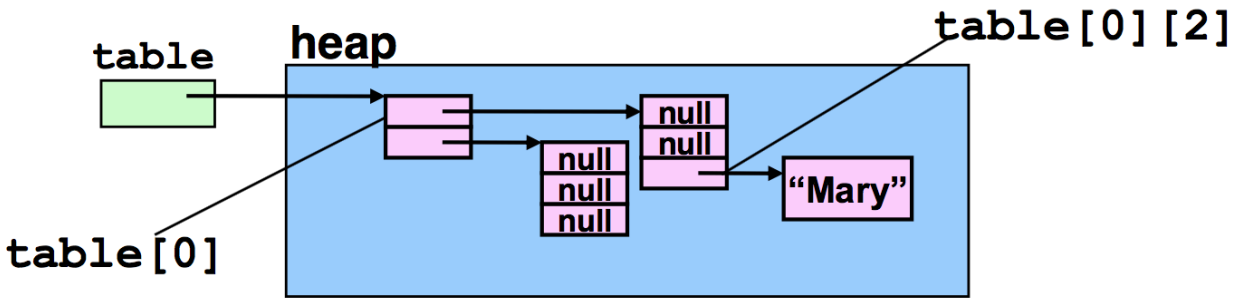
is equivalent to

```
for(int i=0; i<args.length;++i){
   String arg= args[i];
   //...
}
```

Multidimensional array

- Implemented as array of arrays

```
Person[][] table = new Person[2][3];
table[0][2] = new Person("Mary");
```



Rows and columns

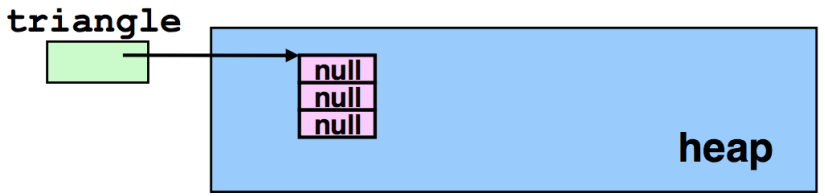
- As rows are not stored in adjacent positions in memory they can be easily exchanged

```
double[][] balance = new double[5][6];
...
double[] temp = balance[i];
balance[i] = balance[j];
balance[j] = temp;
```

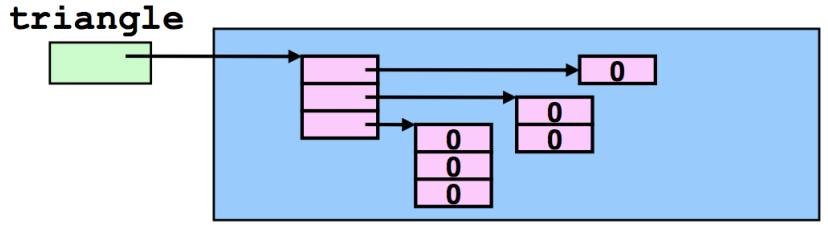
Rows with different length

- A matrix (bidimensional array) is indeed an array of arrays

```
int[][] triangle = new int[3][]
```




```
for (int i=0; i< triangle.length; i++)
    triangle[i] = new int[i+1];
```



Package

Motivation

- Class is a better element of modularization than a procedure. But it is still little
- For the sake of modularization, Java provides the package feature

Package

- A package is a logic set of class definitions
- These classes are made of several files, all stored in the same directory
- Each package defines a new scope (i.e., it puts bounds to visibility of names)
- It‟s then possible to use same class names in different package without name-conflicts

Package name

- A package is identified by a name with a hierarchic structure (fully qualified name)
  - E.g. java.lang (String, System, ...)
- Conventions to create unique names Internet name in reverse order
  - **it.unimo.myPackage**

Example

- java.awt
  - Window
  - Button
  - Menu
- java.awt.event (sub-package)
  - MouseEvent
  - KeyEvent

Creation and usage

- Creation:
- Package statement at the beginning of class file
  - **package packageName;**
- Usage:
- Import statement at the beginning of class file (where needed)
  - **import packageName.className;**
  - **import java.awt.*;**

Access to a class in a package

- Referring to a method/class of a package

int i = myPackage.Console.readInt()

- If two packages define a class with the same name, they cannot be both imported. If you need both classes you have to use one of them with its fully-qualified name:

```
import java.sql.Date;
Date d1; // java.sql.Date
java.util.Date d2 = new java.util.Date();
```

Package and scope

- Scope rules also apply to packages
- The “interface” of a package is the set of public classes contained in the package
- Hints
  - Consider a package as an entity of modularization
  - Minimize the number of classes, attributes, methods visible outside the package

Visibility w/ multiple packages

- public class A { }
  - Public methods/attributes of A are visible outside the package
- class B { }
  - No method/attribute of B is visible outside the package

Access Rules

	Method in the same class	Method of other class in the same package	Method of other class in other package
Private attribute/ method	Yes	No	No
Package attribute / method	Yes	Yes	No
Public attribute / method on package class	Yes	Yes	No
Public attribute / method on public class	Yes	Yes	Yes

Static attributes and methods

Class variables

- Represent properties which are common to all instances of an object
- But they exist even when no object has been instantiated
- They are defined with the static modifier
- Access: *ClassName.attribute*

```
Class car {
    static int numberOfWheels = 4;
}
int y = Car.numberOfWheels;
```

Static methods

- Static methods are not related to any instance
- They are defined with the static modifier
- Access: ClassName.method()

```
Double cos = Math.cos(degree)
```

Memory management

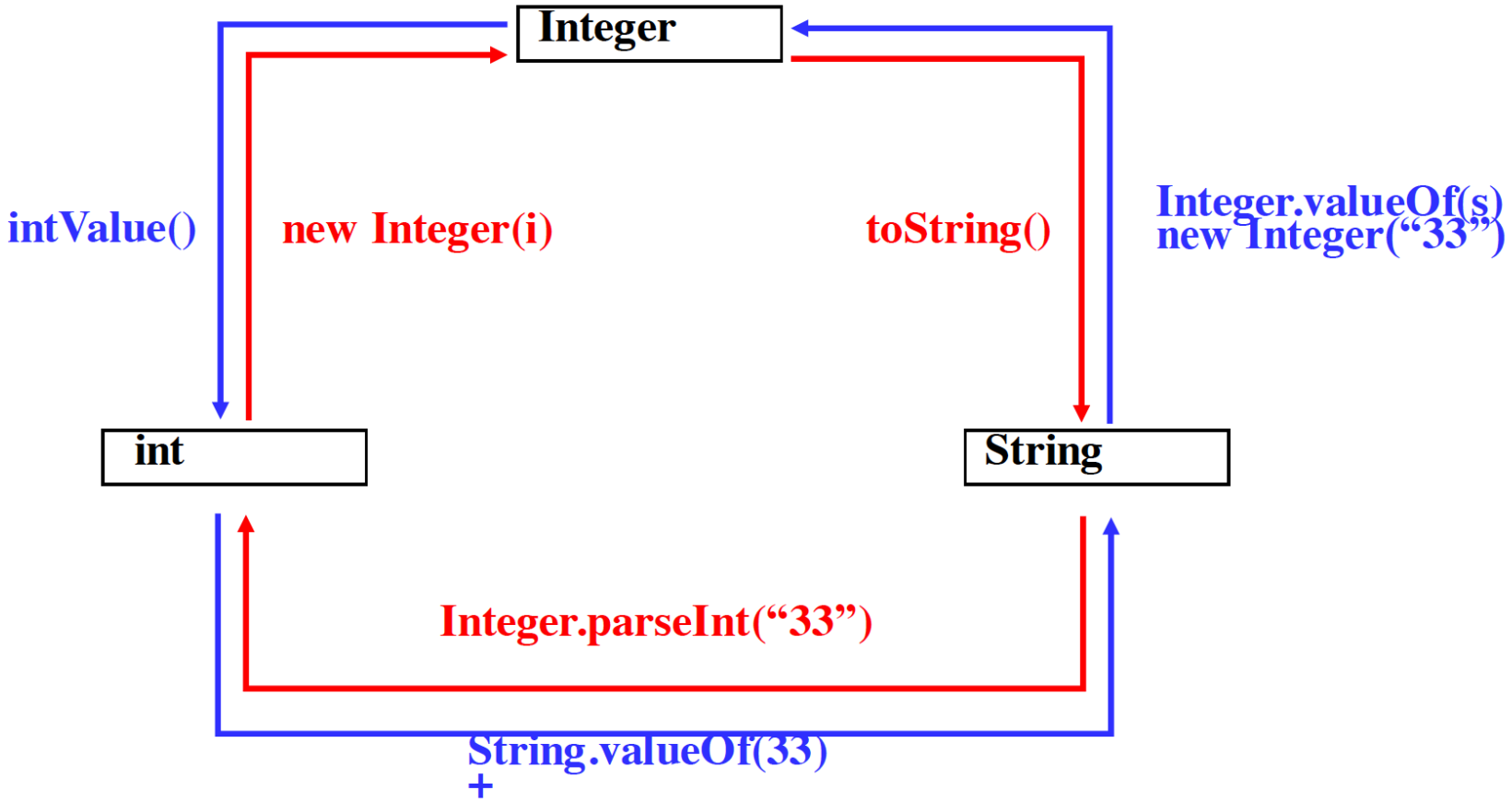
Motivation

- In an ideal OO world, there are only classes and objects
- For the sake of efficiency, Java use primitive types (int, float, etc.)
- Wrapper classes are object versions of the primitive types
- They define conversion operations between different types

Wrapper Classes

Primitive type	Wrapper Class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
void	Void

Conversions



Conversions

```
Integer obj = new Integer(88);
String s = obj.toString();
int i = obj.intValue();
int j = Integer.parseInt("99");
int k = (new Integer(99)).intValue();
```

Autoboxing

- Since Java 5 an automatic conversion between primitive types and wrapper classes (autoboxing) is performed.

```
Integer i = new Integer(2); int j;
j = i + 5;
//instead of:
j = i.intValue() + 5;

i = j + 2;
//instead of:
i = new Integer(j+2);
```