Introduction to Java

Università di Modena e Reggio Emilia

Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

ING

Java Timeline

- 1991: SUN develops a programming language for cable TV set-top boxes
- 1996: Java 1
- 1996: Netscape supports Java. Popularity grows
- 1998: Java 2 (libraries)
- 2005: Java 5 (major enhancements)
- 2014: Java 8

https://en.wikipedia.org/wiki/Java_version_history

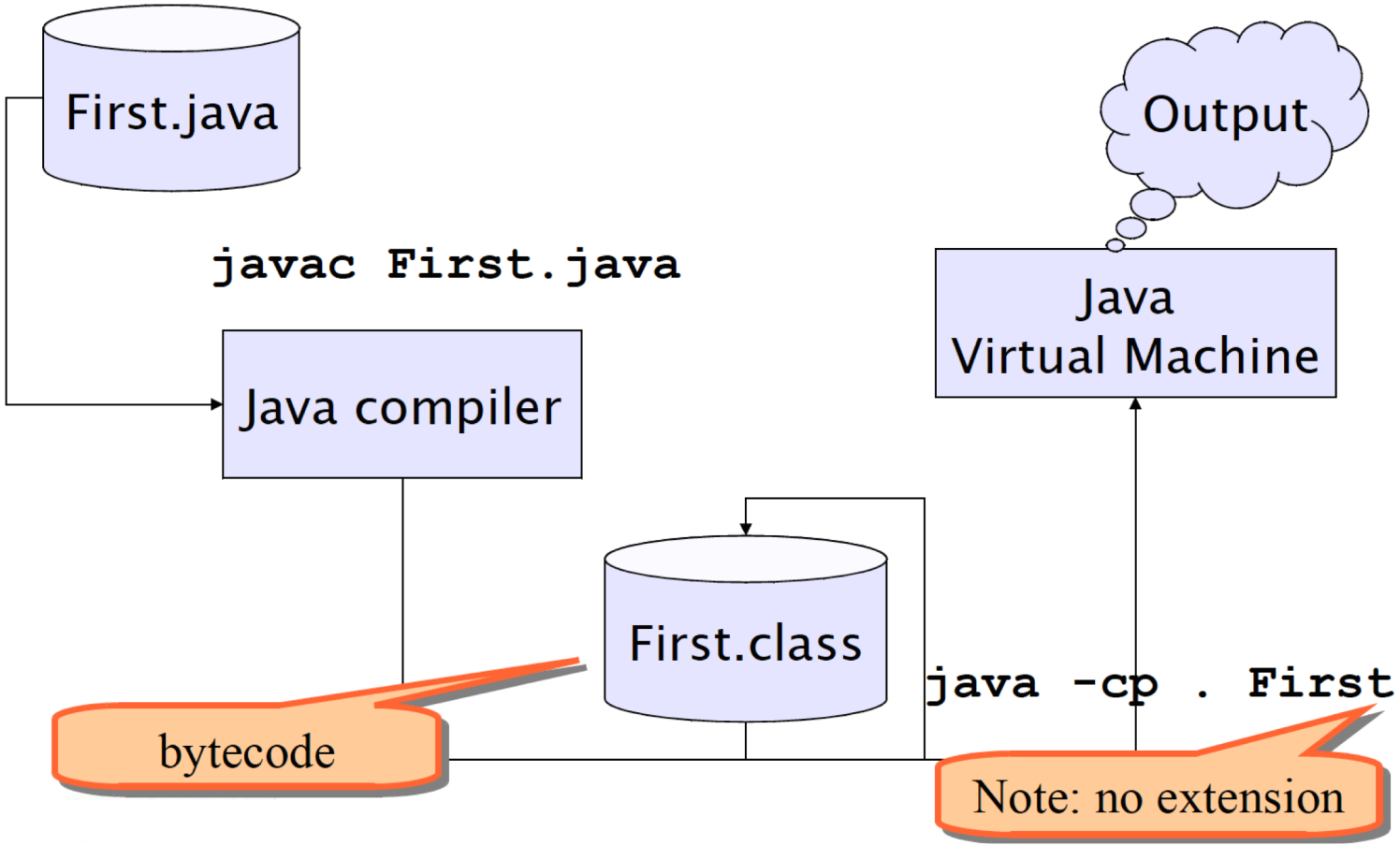
Java Features

- Platform independence (portability)
  - Write once, run everywhere
  - Translated to intermediate language (bytecode)
- Pure OO language
- Strong type model and no pointers
- Exceptions as a pervasive mechanism
- Shares many syntax elements w/ C++ (learning curve less steep)
- Automatic garbage collection
- Run time loading and linking

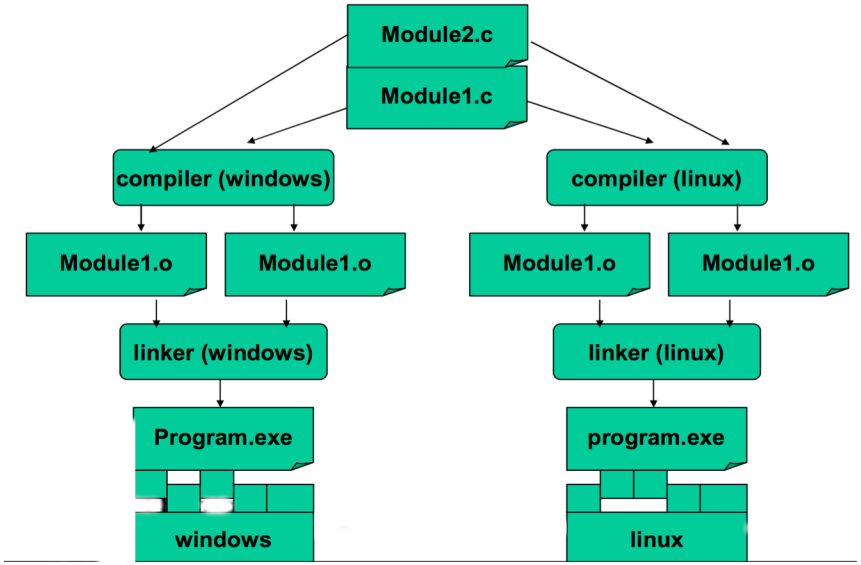
Java Programs

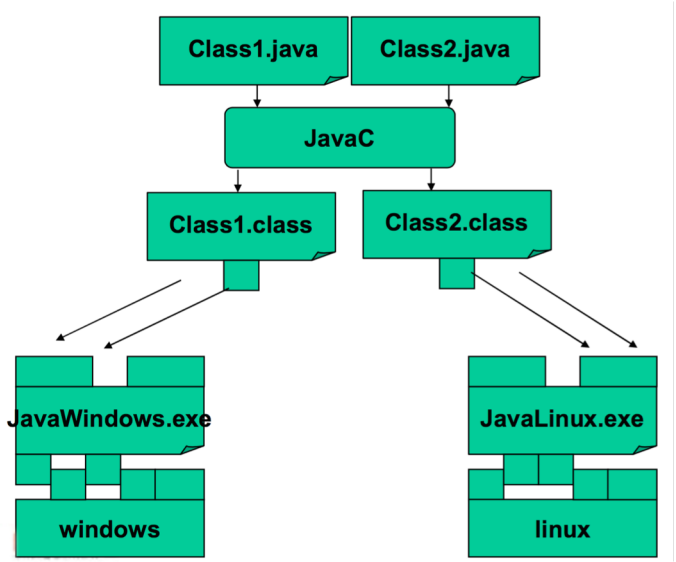
- Application
  - It's a common program, similar to C/C++
  - Runs through the Java interpreter (Java) of the installed Java Virtual Machine (JVM)
- Servlet (Web server)
  - Java program that extends the capabilities of a Web server
  - Java counterpart to other dynamic Web content technologies such as PHP and ASP.NET

Building and running



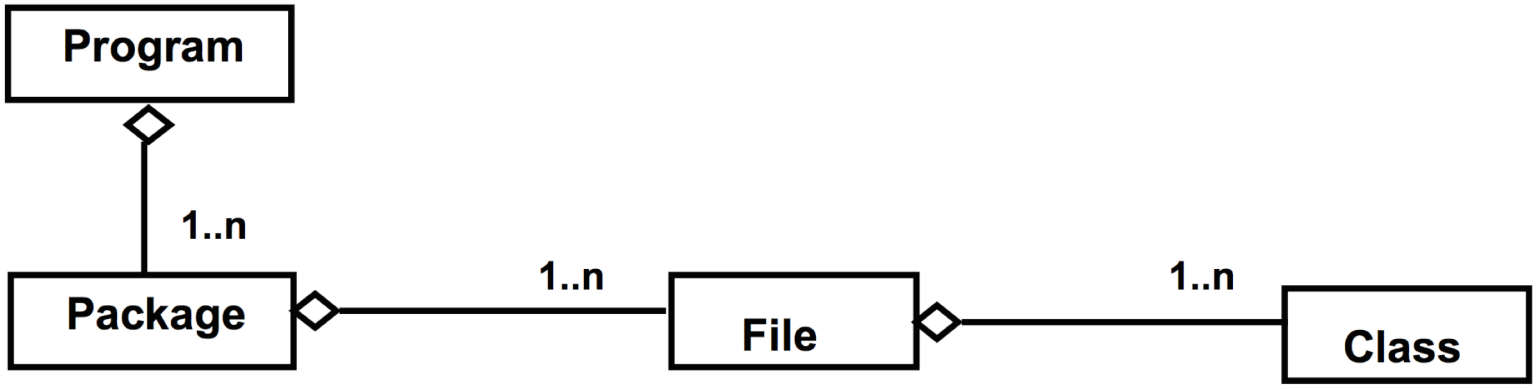
Compiled vs Interpreted





Program, files and classes

- A program is made of one or more packages, containing one or more files
- A file contains one *public* class and, optionally, multiple *private* classes. The file name must be equal to the public class name.



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) {
    …
}
```

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

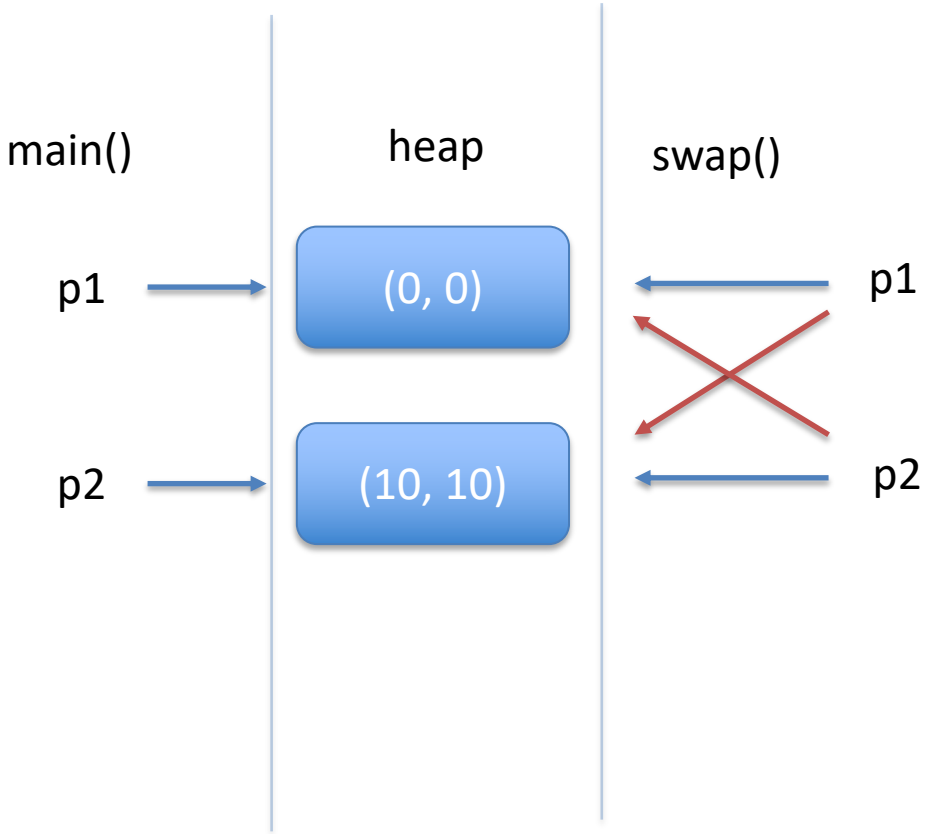
- Same as C
  - if-else
  - switch-case
  - while
  - do-while
  - for
  - break
  - continue

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

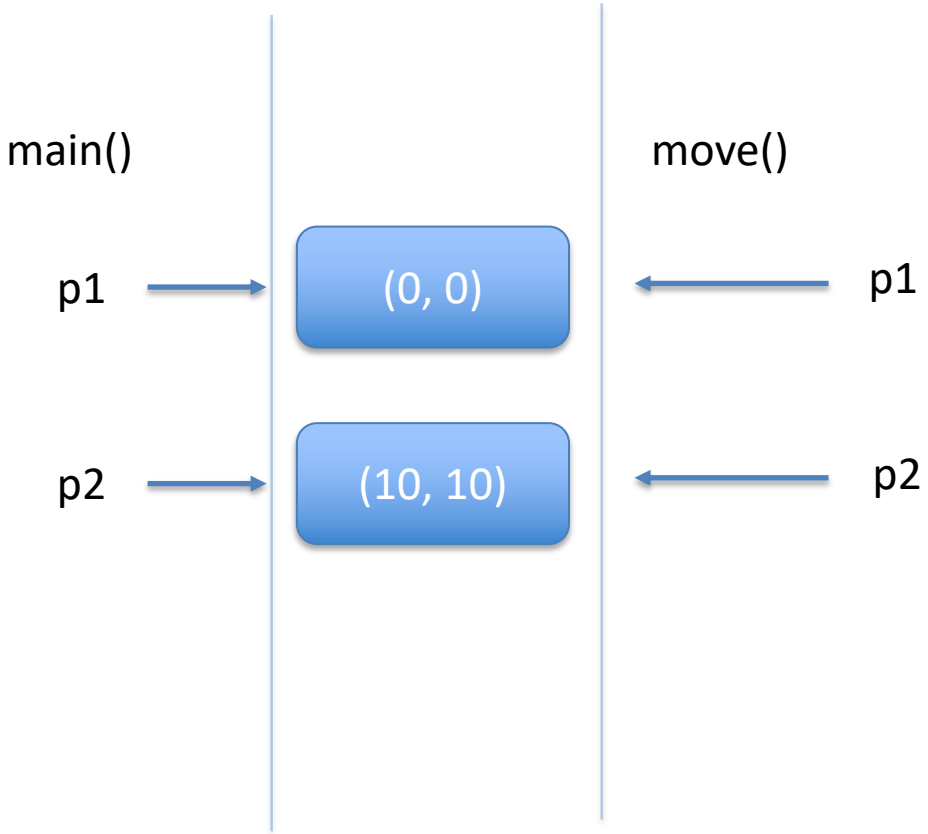
Passing Parameters

```
public class Parameters {
  public static void main(String[] args) {
    Point p1 = new Point(0, 0);
    Point p2 = new Point(10, 10);
    System.out.println(p1);
    System.out.println(p2);

    swap(p1, p2);

    System.out.println(p1);  // 0, 0
    System.out.println(p2);  // 10, 10
  }


  public static void swap(Point p1, Point p2) {
    Point tmp = p1;
    p1 = p2;
    p2 = tmp;
  }
}
```



Passing Parameters

```
public class Parameters {
  public static void main(String[] args) {
    Point p1 = new Point(0, 0);
    Point p2 = new Point(10, 10);
    System.out.println(p1);
    System.out.println(p2);

    move(p1, p2);

    System.out.println(p1);  // 10, 10
    System.out.println(p2);  // 0, 0
  }


  public static void move(Point p1, Point p2) {
    p1.move(10, 10);
    p2.move(0, 0);
  }
}
```



Primitive types

Category	Types	Size (bits)	Minimum Value	Maximum Value	Precision	Example
Integer	`byte`	8	-128	127	From +127 to -128	`byte b = 65;`
	`char`	16	0	2^{16}-1	All Unicode characters[1]	`char c = 'A'; char c = 65;`
	`short`	16	-2^{15}	2^{15}-1	From +32,767 to -32,768	`short s = 65;`
	`int`	32	-2^{31}	2^{31}-1	From +2,147,483,647 to -2,147,483,648	`int i = 65;`
	`long`	64	-2^{63}	2^{63}-1	From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808	`long l = 65L;`
Floating-point	`float`	32	2^{-149}	$(2-2^{-23})\cdot 2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	`float f = 65f;`
	`double`	64	2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	From 1.797,693,134,862,315,7 E+308 to 4.9 E-324	`double d = 65.55;`
Other	`boolean`	--	--	--	false, true	`boolean b = true;`
	`void`	--	--	--	--	--

Constants

- The final modifier

```
final float PI = 3.1415;
PI = 16.0;   // ERROR, no changes allowed
```

- Use uppercases (coding conventions)

Operators (integer and floating-point)

- Operators follow C syntax:
  - arithmetical + - * / %
  - Relational == != > < >= <=
  - bitwise & | ^ ! >> <<
  - Logical && || ! ^
  - Assignment = += -= *= /= %= &= |= ^=
  - Increment ++ --
- Chars can be treated as integers (e.g. switch)
- Logical operators work ONLY on booleans. int is NOT considered a boolean value like in C

Switch with chars

```
public class Test {
   public static void main(String args[]) {
       char grade = ‘A';

      switch(grade) {
         case 'A' :
            System.out.println("Excellent!");
            break;
         case 'B' :
         case 'C' :
            System.out.println("Well done");
            break;
         case 'D' :
            System.out.println("You passed");
         case 'F' :
            System.out.println("Better try again");
            break;
         default :
            System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
   }
}
```

Coding Conventions

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

In Eclipse: CTRL+A, CTRL-I **(Auto Indent)**

Strings

String

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

String

Strings are immutable. the JVM can optimize the amount of memory allocated for them by storing only one copy of each literal String in the pool. This process is called interning.

When we create a String variable and assign a value to it, the JVM searches the pool for a String of equal value. If found, the Java compiler will simply return a reference to its memory address, without allocating additional memory.

If not found, it'll be added to the pool (interned) and its reference will be returned.

```
String first = "Baeldung";
String second = "Baeldung";
System.out.println(first == second);  // True
```

String

Let's create two different objects using *new* and check that they have different references:

```
String first = new String("Baeldung");
String second = new String("Baeldung");
String third = "Baeldung";

System.out.println(first == second);  // False
System.out.println(first == third);   // False
```

String

- String: an object storing a sequence of text characters.
- Creating a string:

```
String name = "text";
String name = expression;
```

- Examples:

```
String name = "Marty Stepp";
int x = 3;
int y = 5;
String point = "(" + x + ", " + y + ")";
```

String

Method name	Description
`int length()`	number of characters in this string
`String substring(int from, int to)`	returns the substring beginning at `from` and ending at `to-1`
`String substring(int from)`	returns `substring(from, length())`
`int indexOf(String str)`	returns the index of the first occurrence of `str`; returns `-1` if not found
`int compareTo(String other)`	returns a value < 0 if `this` is less than `other` returns a value = 0 if `this` is equal to `other` returns a value > 0 if `this` is greater than `other`

The *equals* method

- Objects (not only Strings) are compared using a method named equals.

```
String s1,s2;

if (s1 == s2) {
    System.out.println(”s1 and s2 point to the same String
    object");
}

if (s1.equals(s2)) {
    System.out.println("The strings are equal!");
}
```

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

- Fast way 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(),…)*

StringBuffer

```
// string concatenation using +
String str = new String (“GNU is not  ”);
str += “Unix”;

// string concatenation using stringbuffer
// each concatenation does not create a new String object
StringBuffer str = new StringBuffer ("Stanford ");
str.append("Lost!!");
```

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 (no actual objects!)
- Array size must be defined at creation time (cannot change afterwards)

Array declaration

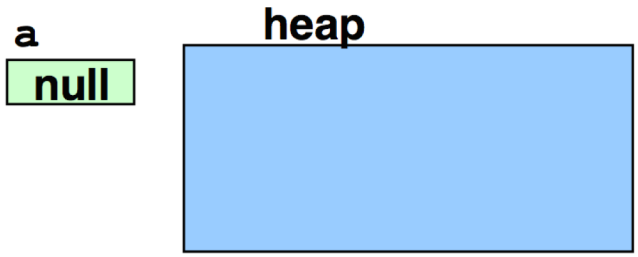
- An array reference can be declared with one of these equivalent syntaxes
  - int[] v, int v[]
  - Point[] v, Point v[]
- An array is an Object, must be created using new, and it is stored in the heap (as all objects)
- Array declaration (e.g., int[] v) allocates memory space for a reference, whose default value is null

Array creation

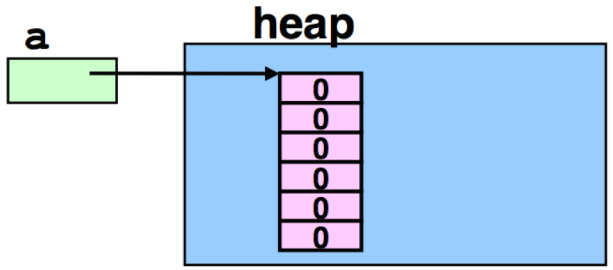
- Using the new operator
  - int[] v = new int[256];
- Using static initialization, filling the array with values
  - int[] v = {2,3,5,7,11,13};

Example – Primitive types

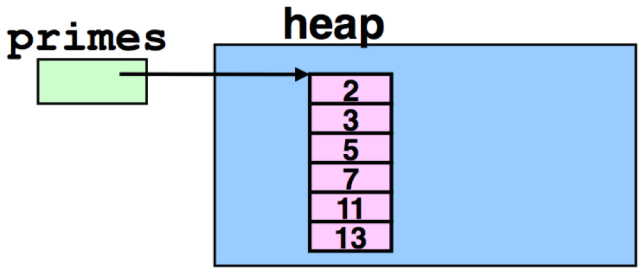
```
int[] a;
```




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

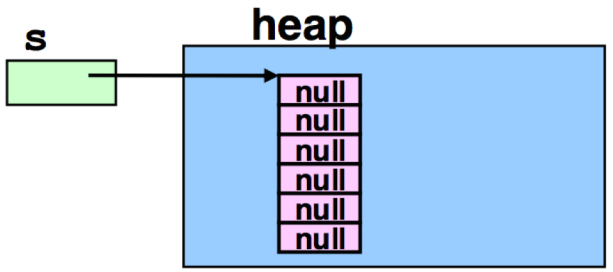



```
int[] primes =
  {2,3,5,7,11,13};
```

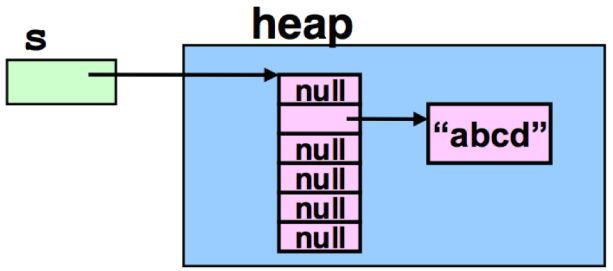


Example – Object reference

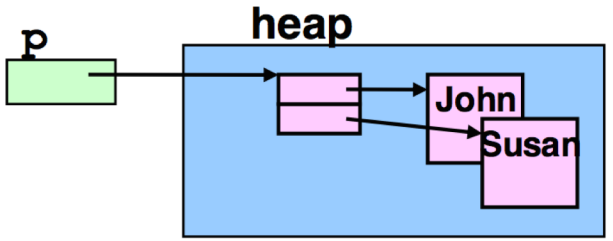
```
String[] s = new
  String[6];
```




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




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



Operations on arrays

Java checks array bounds

```
int[] v = new int[16]
System.out.println(v[20])
```

ArrayIndexOutOfBoundsException (runtime!)

Array length (size of the array, not number of contained elements) is given by the attribute length

```
for (int i=0; i < v.length; i++) {
  v[i] = i;
}
```

Operations on arrays

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

Operations on arrays

- New loop construct:
  `for( Type var : set_expression )`
- *set_expression* can be either
  - an array
  - a class implementing Iterable (see Java Collection Framework)
- The compiler can automatically loop with correct indexes
  - less error prone

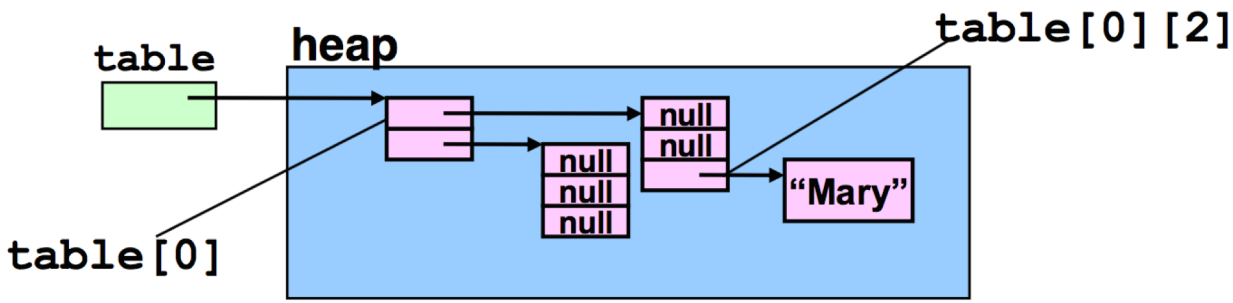
For each

```
String[] args;

/* implicit index */
for(String arg: args){
    System.out.println(arg);
    //...
}

/* explicit index */
for(int i = 0; i < args.length; i++){
    System.out.println(args[i]);
    //...
}
```

Multidimensional array

- Implemented as array of arrays

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



Rows and columns

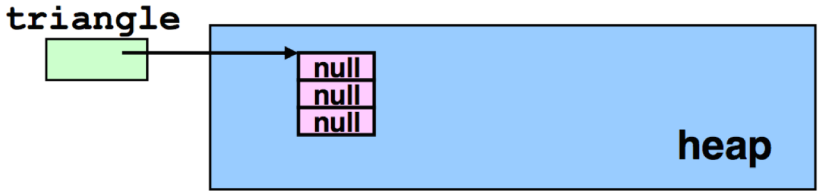
- As rows are not stored in adjacent positions in memory they can be easily exchanged

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

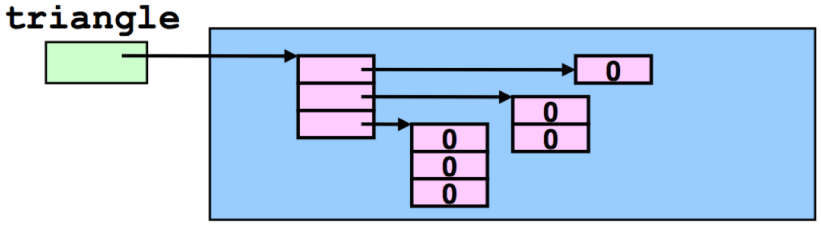
Rows with different length

- 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];
```

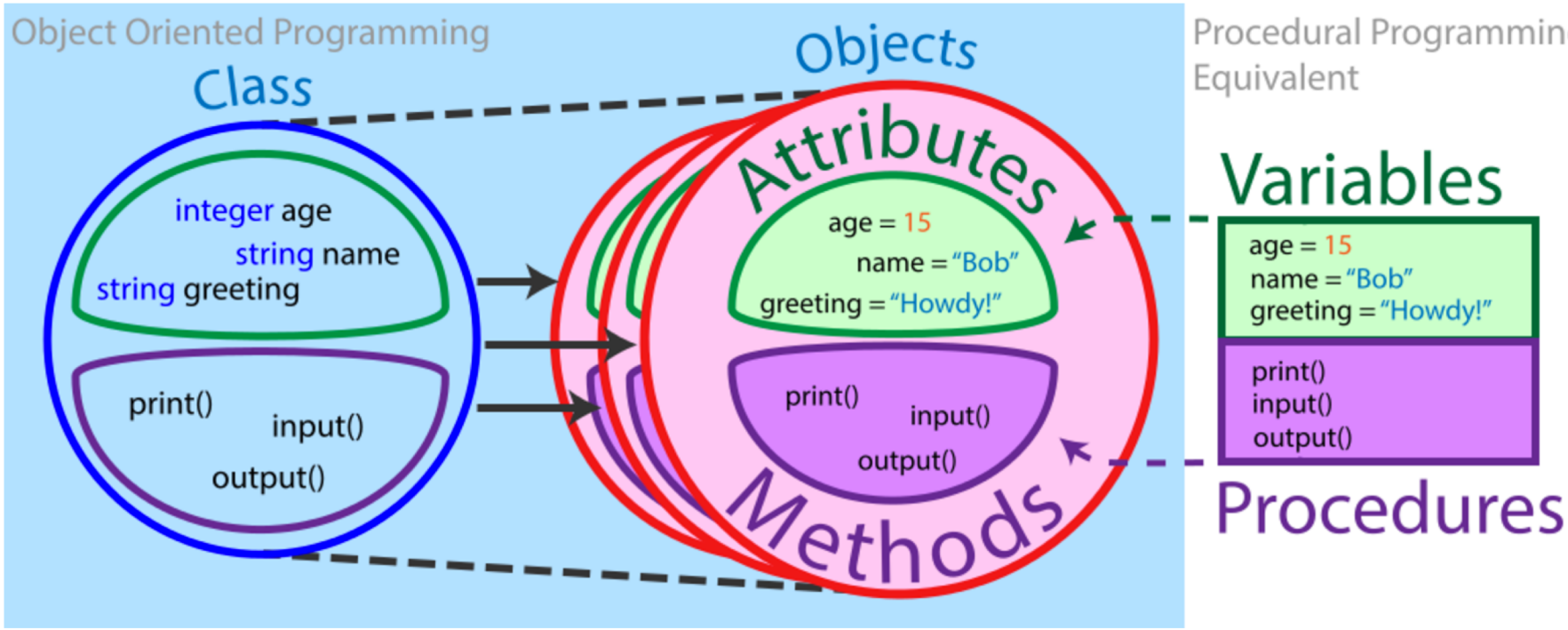


Classes and Objects

Class

Descriptor of a class of objects *(Platonic idea)*

Class


Object Oriented Programming
Class
integer age
string name
string greeting
print()
input()
output()
Objects
Attributes
age = 15
name = "Bob"
greeting = "Howdy!"
print()
input()
output()
Methods
Procedural Programming Equivalent
Variables
age = 15
name = "Bob"
greeting = "Howdy!"
print()
input()
output()
Procedures

Class Definition

```
public class Car {
  public boolean isOn;
  public String brand;
  public String color;

  public String getColor() {
    return color;
  }

  public void setColor(String color) {
    this.color = color;
  }

  public String toString() {
    return "This is a car!";
  }
}
```

Class name

Attributes

Methods

Information hiding

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

public class App {
    public static void main(String[] args) {
        Car c = new Car();
        c.color = “red”;      /* Works but unsafe! */
    }
}
```

Information hiding

```
public class Car {
    private String color;
    public void setColor(String color) {
        this.color = color;
    }
}

public class App {
    public static void main(String[] args) {
        Car c = new Car();
        c.color = “red”;        /* Compiler error */
        c.setColor(“red”);      /* Works, Safe! */
    }
}
```

Visibility

Most Restrictive ⟷ Least Restrictive

Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Constructors

- Constructor method contains operations we want to execute as soon as objects are created (attributes initialization!)
- If a constructor is not defined within a class, a default one (with no parameters) is defined.

Getters and Setters

- Since attributes are usually encapsulated, methods for reading and writing them are frequently useful. These methods are called getters and setters.
- In Eclipse you can generate them automatically!
- It is worth noting that, for reducing the number of errors, using the same name for method parameters and class attributes is a good practice!

```
class Car {
    String color;
    ...
    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

toString()

- It is handy to obtain a textual representation from objects.
- In order to provide objects with this feature the method *String toString()* have to be implemented.

Method Overloading

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

Method Overloading

```
public class Foo {
    public int doit(int n, String s) {
        return n + s.length();
    }
    public int doit(String s, int n) {
        return n * s.length();
    }
    public static void main(String[] args) {
        Foo f = new Foo();
        System.out.println(f.doit(5, "Foo"));  // 8
        System.out.println(f.doit("Foo", 5));  // 15
    }
}
```

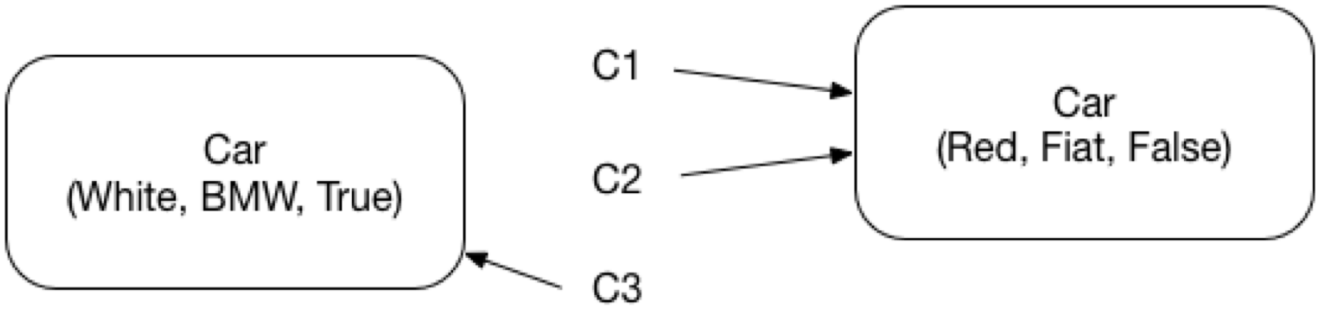
Object definition

- An object is identified by:
  - Its class, which defines its structure in terms of attributes and methods
  - Its state (attributes values)
- An internal unique identifier
  - *try: System.out.println(new int[16]);*
- Zero, one or more reference can point to the same object

Object creation

- Creation of an object is made with the keyword new
- It returns a reference to the piece of memory containing the created object

```
Car c1 = new Car(Red, Fiat, False);
Car c2 = c1;
Car c3 = new Car(White, BMW, True)
```



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
- Constructors can have parameters
  - `String s = new String();`
  - `String s = new String("ABC");`

Constructors

- Constructor method contains operations we want to execute as soon as objects are created (attributes initialization!)
- Overloading of constructors is often used
- If a constructor is not defined within a class, a default one (with no parameters) is defined.
- If a constructor is defined, the default one is disabled!

Constructors

```
public class Car {
     private String color;
     public void setColor(String color) {
          this.color = color;
     }
}

public class App {
     public static void main(String[] args) {
          /* Works with default constructor!
          Possibly unsafe: attributes not initialized */
          Car c = new Car();
     }
}
```

Constructors

```
public class Car {
     private String color;
     /* Constructor */
     public Car(String color) {
          this.color = color;
     }
     public void setColor(String color) {
          this.color = color;
     }
}

public class App {
     public static void main(String[] args) {
          Car c = new Car();       /* Error! Default Constructor missing! */
          Car c = new Car("Red"); /* Works with defined constructor! */

     }
}
```

Constructors

```
public class Car {
     private String color;
     public Car() {
          this.color = “Red”;
     }

     public Car(String color) {
          this.color = color;
     }
     public void setColor(String color) {
          this.color = color;
     }
}

public class App {
     public static void main(String[] args) {
          Car c = new Car();       /* Works with defined constructor! */
          Car c = new Car(“Red”); /* Works with defined constructor! */
     }
}
```

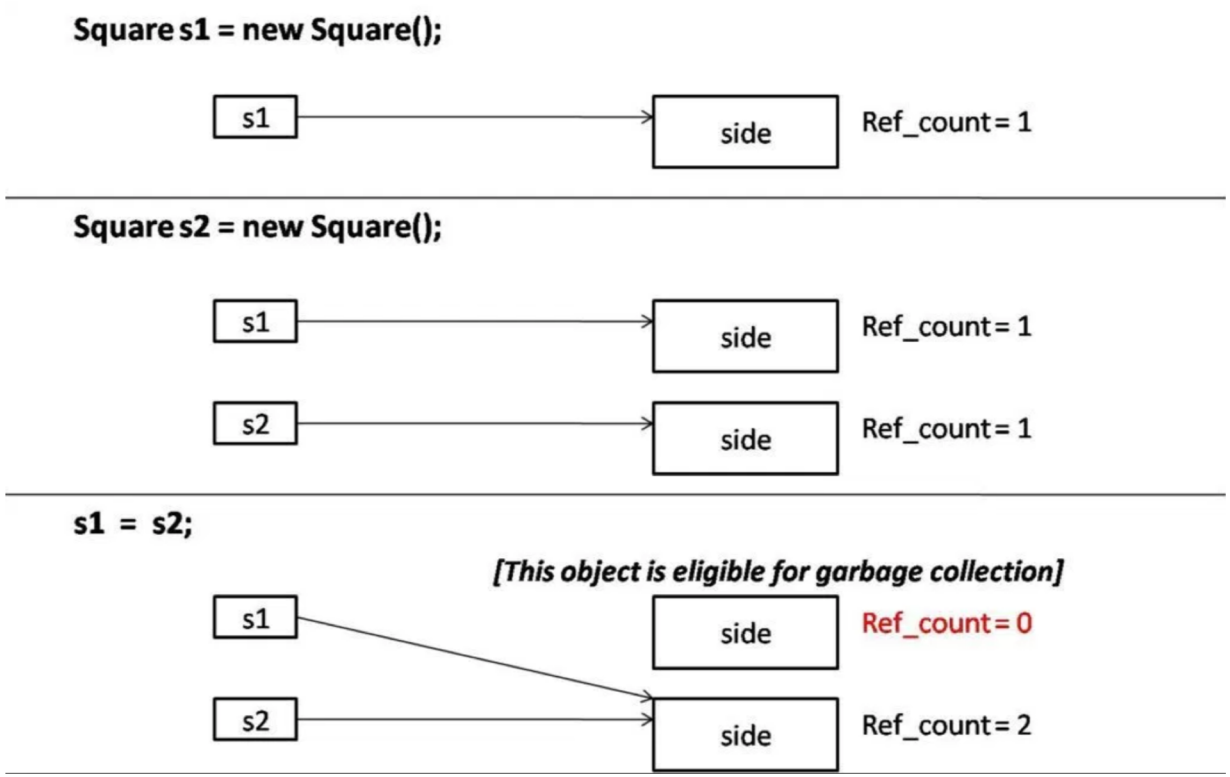
The keyword this

- It can be useful in methods to distinguish between object attributes and local variables (this represents a reference to the current object)
- Accessing attributes or methods of the same object do not need using object reference

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

Objects destruction

- It is no longer a programmer concern
- Java uses *Garbage Collection (an automatic way for de-allocating unreferenced objects)*



Operations on references

- Only the relational operators == and != are defined
  - 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
- There is NO pointer arithmetic

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 class **PrintStream** (representing the standard output)
- **println()** is a method of **PrintStream** which prints a string

Static attributes and methods

Static attributes and methods

- Represent properties (attributes) and behaviors (methods) which are common to all instances of an object
- They exist even when no object has been instantiated!
- They are defined with the static modifier
- Access: *ClassName.attributename|methodname*

Static attributes and methods

```
Class Car {
    static final int nWheels = 4;
    /* … */
}

public static void main(String[] args) {
    /* access to static attributes */
    int nw = Car.nWheels;
    double pi = Math.PI;
    …
    /* access to static methods */
    Double cos = Math.cos();
}
```

Wrapper Classes

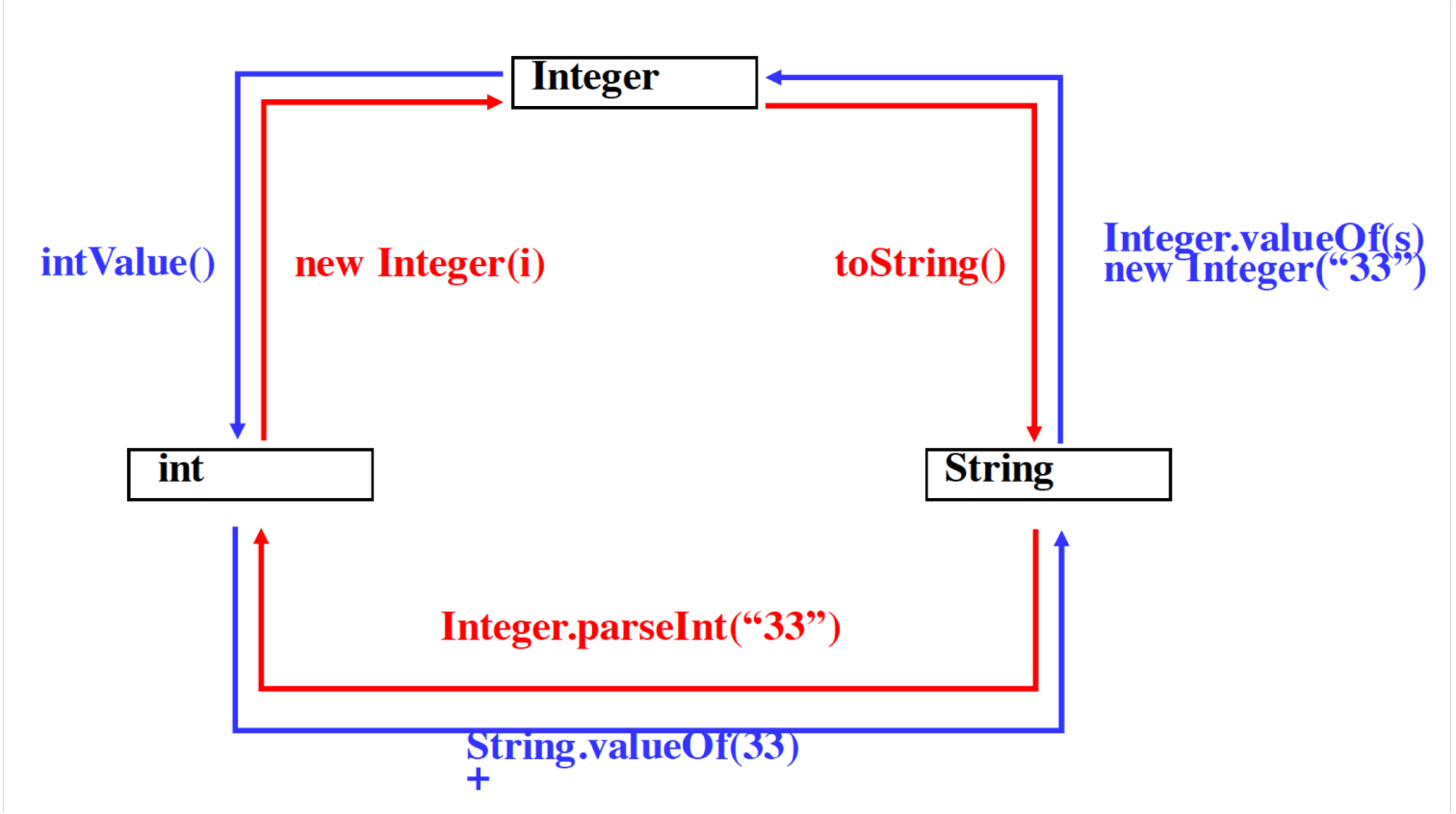
Wrapper Classes

- 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
- Constructors (e.g., new Integer(5)) have been deprecated since Java 9
- They provide conversion operations among Strings, Objects, and primitive 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



Auto boxing

- Auto boxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called Unboxing.

```
class AutoboxingExample {
    public static void myMethod(Integer num){
        System.out.println(num);
    }
    public static void main(String[] args) {
        int i = 2;
        myMethod(i);
    }
}
```

Auto unboxing

- Auto boxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called Unboxing.

```
class UnboxingExample {
    public static void myMethod(int num){
        System.out.println(num);
    }
    public static void main(String[] args) {
        Integer i = new Integer(100);
        myMethod(i);
     }
}
```

Package

Motivation

- Class is a better element of modularization than a procedure. But it is still little (50-250 lines on average)
- For the sake of modularization, Java provides packages

	junit	fitnesse	testNG	tam	jdepend	ant	tomcat
max	500	498	1450	355	668	2168	5457
mean	64.0	77.6	62.7	95.3	128.8	215.9	261.6
min	4	6	4	10	20	3	12
sigma	75	76	110	78	129	261	369
files	90	632	1152	69	55	954	1468
total lines	5756	49063	72273	6575	7085	206001	384026

Package

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

Package names

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

Definition and usage

- Definition: Package statement at the beginning of class file
  - **package packageName;**

- Usage: Import statement at the beginning of class file
  - **import packageName.className;**

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 = new Date(); // 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 from the outside (of the package)