Java type system

Giorgio Brajnik

giorgio.brajnik@uniud.it

Oct 2021

Java specification and documentation

Java 8 specification:

- http://docs.oracle.com/javase/8/docs/api/overview-summary.html
- eg. http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html

Stack overflow:

- http://stackoverflow.com/questions/5131131/what-happens-when-you-increment-an-integer-beyond-its-max-value

Tutorial

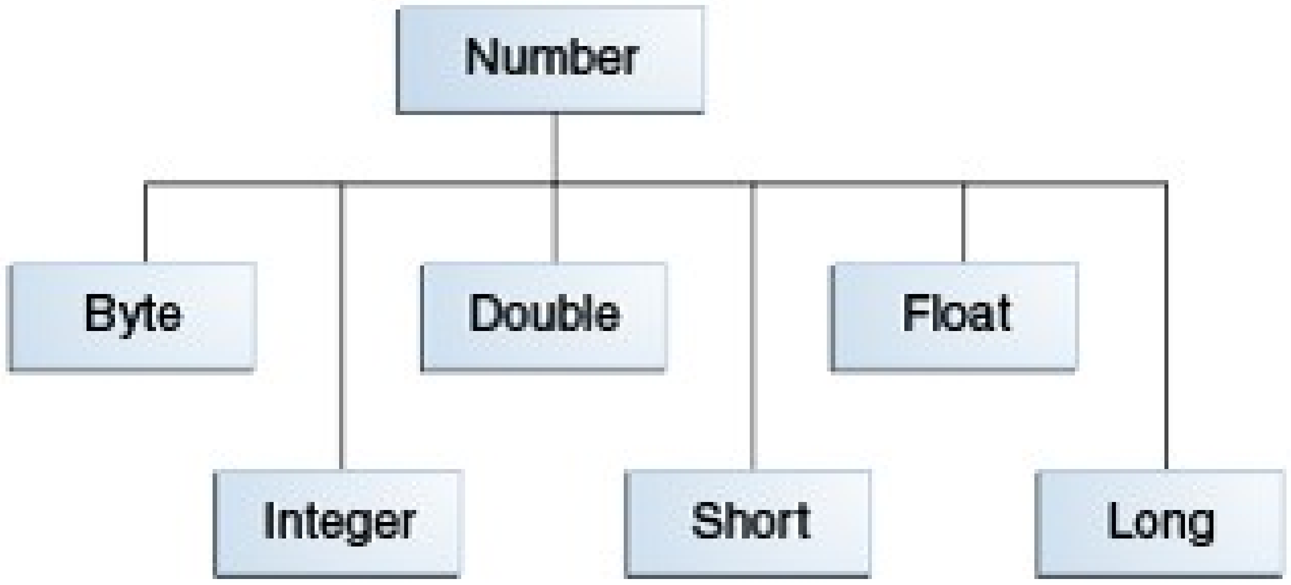
- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Primitive types

- **byte**: 8-bit signed 2 complement integer
- **char**: 16-bit Unicode
- **int**: 32-bit signed 2 complement integer
- **short:** 16-bit signed 2 complement integer
- **long**: 64-bit signed 2 complement integer
- **float**: 32-bit floating point
- **double**: 64-bit floating point
- **boolean**: true/false

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Numbers classes



Boxing:

```
int i = 33;
Integer x = new Integer(i);
// or
Integer x = Numbers.valueOf(i);
```

Unboxing:

```
int i = 33;
Integer x = new Integer(i);
int j = x.intValue();
```

https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html

Number/Integer/... vs primitive types

- some method expects `Object`
- to be able use constants defined in the class:
  - Integer.MAX_VALUE
- to be able use conversion methods:
  - between Numbers
  - to/from strings
  - changing base representations

Conversions

```
Number:
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()

static Integer decode(String s)
static int parseInt(String s)
static int parseInt(String s, int radix)
String toString()
static String toString(int i)
static Integer valueOf(int i)
static Integer valueOf(String s)
static Integer valueOf(String s, int radix)
```

Autoboxing

Automatic type conversions

```
Character ch = 'a';
```

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2){
    li.add(i);
}
return(li);
```

… is equivalent to

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2){
    li.add(Integer.valueOf(i)); // oppure li.add( new Integer(i) )
}
return(li);
```

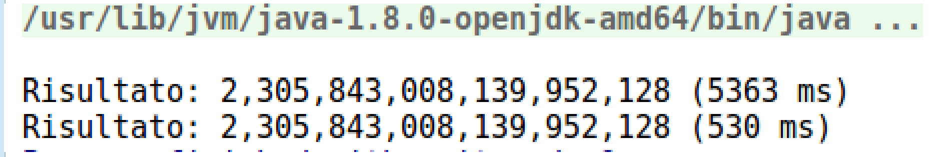
Autoboxing may be dangerous

```java
/**
 * classe per mostrare esempio di inefficienza
 * dovuto al modo con cui si fa autoboxing e quindi il numero di volte che si
 * istanzia un oggetto.
 * In sum1() ogni i viene convertito in Long prima di essere sommato.
 */
public class SumOfIntWithLong {
    public static void main(String[] args) {
        long res = 0;
        long before = System.currentTimeMillis();
        res = sum1();
        long after = System.currentTimeMillis();
        System.out.format( "%nRisultato: %,d (%d ms)", res, (after - before) );
        res = 0;
        before = System.currentTimeMillis();
        res = sum2();
        after = System.currentTimeMillis();
        System.out.format( "%nRisultato: %,d (%d ms)", res, (after - before) );
    }

    private static long sum1() {
        Long sum = 0L;
        for( long i = 0; i <= Integer.MAX_VALUE; i++){
            sum += i; // <-- autoboxing critico
        }
        return sum;
    }

    private static long sum2() {
        long sum = 0L;
        for( long i = 0; i <= Integer.MAX_VALUE; i++){
            sum += i;
        }
        return sum;
    }
}
```

Execution:



```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...

Risultato: 2,305,843,008,139,952,128 (5363 ms)
Risultato: 2,305,843,008,139,952,128 (530 ms)
```

Characters and Strings

char and Character

```
char ch = 'a';
char omega = '\u03A9'; // unicode
char[] charArray = { 'a', 'b', 'c', 'd', 'e' };
Character ch = new Character('a');
```

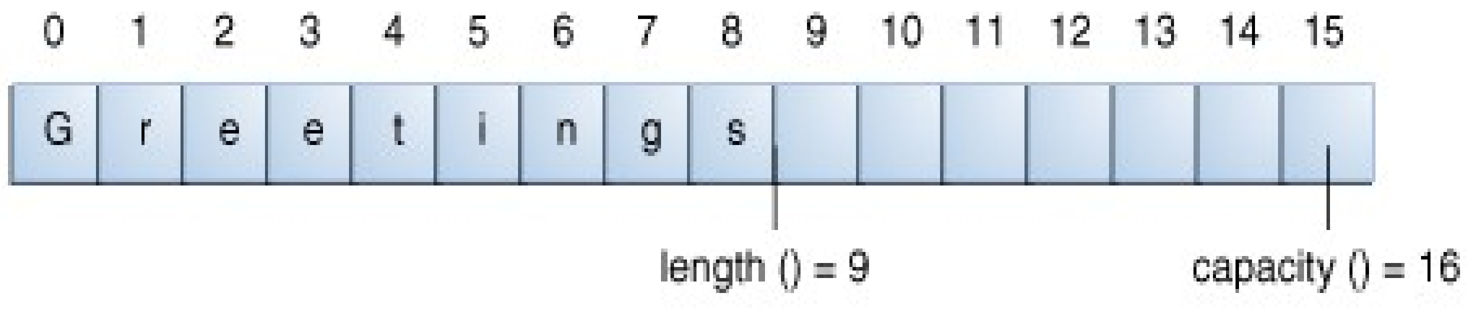
Strings

```
String greeting = "Hello world!";
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
int len = helloString.length();
helloString.isEmpty();
String s = "Hello," + " world" + "!";
```

Characters and String

StringBuilder

```
StringBuilder sb = new StringBuilder();
sb.append("Greetings");
```



String vs StringBuilder

- String is immutable
- StringBuilder is mutable
  - https://docs.oracle.com/javase/tutorial/java/data/buffers.html

Printing

- System.out is instance of java.io.PrintStream
- it has methods such as:
  - print, println, printf, format

Output formatting

```
  long n = 461012;
System.out.format("%d%n", n);      //  -->  "461012"
System.out.format("%08d%n", n);    //  -->  "00461012"
System.out.format("%+8d%n", n);    //  -->  " +461012"
System.out.format("%,8d%n", n);    // -->  " 461,012"
System.out.format("%+,8d%n%n", n); //  -->  "+461,012"
double pi = Math.PI;
System.out.format("%f%n", pi);       // -->  "3.141593"
System.out.format("%.3f%n", pi);     // -->  "3.142"
System.out.format("%10.3f%n", pi);   // -->  "     3.142"
System.out.format("%-10.3f%n", pi);  // -->  "3.142     "
System.out.format(Locale.ITALY,
                  "%-10.4f%n%n", pi); // -->  "3,1416    "
Calendar c = Calendar.getInstance();
System.out.format("%tB %te, %tY%n", c, c, c); // -->  "May 29, 2006"

System.out.format("%tl:%tM %tp%n", c, c, c);  // -->  "2:34 am"

System.out.format("%tD%n", c);    // -->  "05/29/06"
```

https://docs.oracle.com/javase/tutorial/java/data/numberformat.html

Enumeration

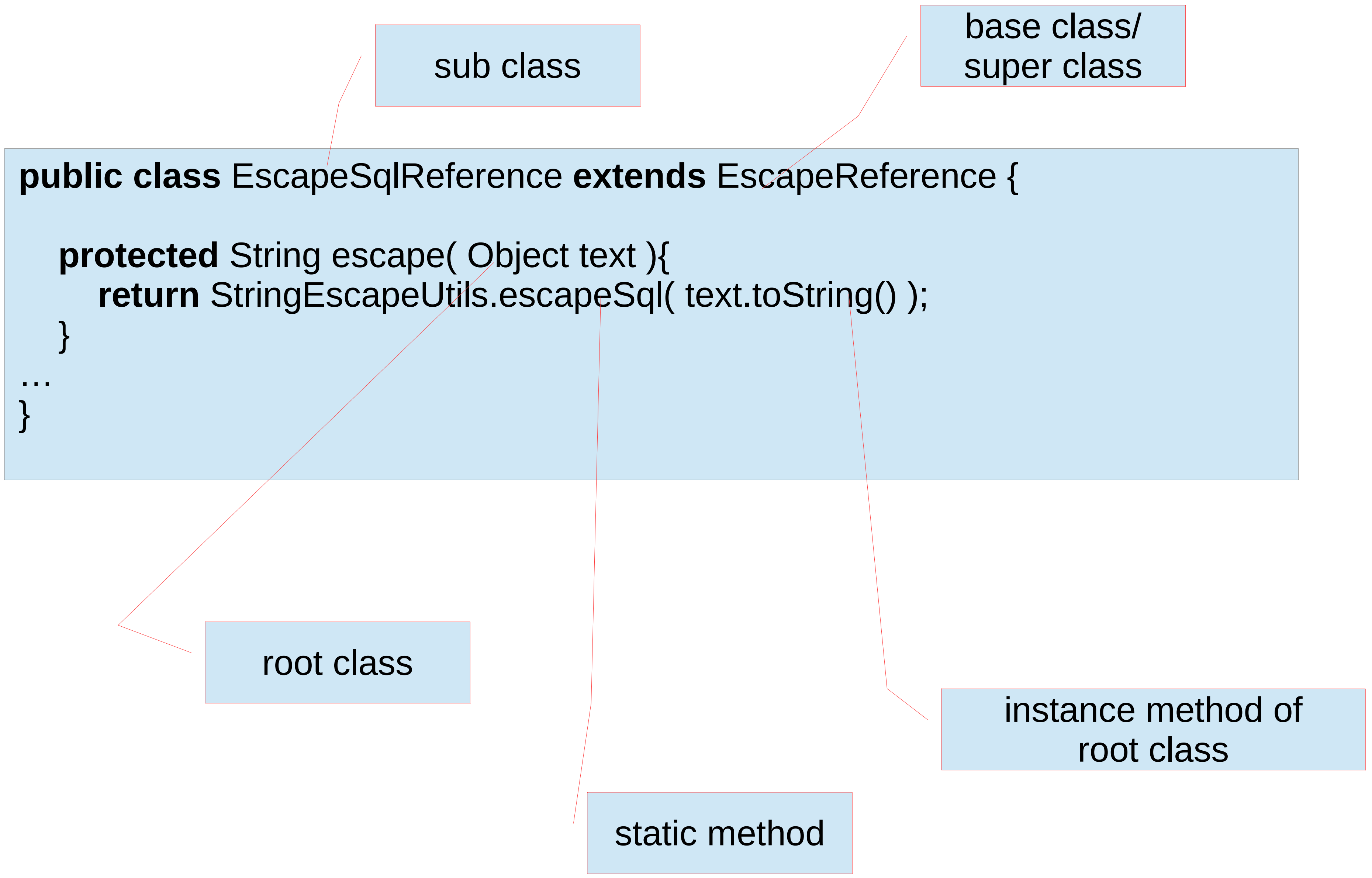
```
public enum MyDay {
           SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
           THURSDAY, FRIDAY, SATURDAY
       }…
    DayOfWeek firstDay = new DayOfWeek(MyDay.MONDAY);
        firstDay.tellItLikeItIs();
    DayOfWeek thirdDay = new DayOfWeek(MyDay.WEDNESDAY);
        thirdDay.tellItLikeItIs();
...
      switch (day) {
           case MONDAY:
                System.out.println("Mondays are bad.");
                break;
            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

                     default:
                            System.out.println("Wrong day.");
                     break;
```

Type checking

- Java is **Strongly Statically Typed**
  - every variable has a type
  - every method has its signature
  - compiler checks type correctness of each assignment (and call)
    - eg int[] doInsertionSort(int[] a)
- compiler deduces the **apparent type** of an expression
  - at COMPILE TIME
  - eg. List<Student> theBestOnes = new ArrayList<Student>();
    - List<Student>: apparent type of theBestOnes
    - ArrayList<Student>: real type of theBestOnes
- JVM manipulates the **real type**
  - at RUN TIME
- compiler and JVM ensure **type safety**:
  - no errors due to type mismatch can occur
  - no out-of-bounds errors can occur
  - no dangling references can occur

Type Hierarchy



```
public class EscapeSqlReference extends EscapeReference {

    protected String escape( Object text ){
        return StringEscapeUtils.escapeSql( text.toString() );
    }
…
}
```

Type Hierarchy

- if T extends S:
  - T MUST HAVE all methods of S
- the **actual type** of an expression is a subtype of its **apparent type**

```
int [] a = new int[3];
Object x = a;
```

apparent type(a) → int[]
real type(a) → int[]

apparent type(x) → Object
real type(x) → int[]

Type Hierarchy

```
int [] a = new int[3];
Object x = a;


a = x; // ILLEGAL!

a.length // → 3

x.length // ILLEGAL!

((int[]) x).length // → 3 (type casting)
```

Overloading

```
public class C {
        public double power(int i, long c){ // do AAA }
        public double power(long x, int c){ // do BBB }
        public double power(long x, long y) { // do CCC }
}       …
        C cc = new C();
        int x;
        long y;
        double z;
        cc.power(x, y) // what does it do?
                → AAA (it is the most specific)
        cc.power(x, x)
                → ERROR because there is no “most specific”
        cc.power((long x), x)
                → BBB
```

method dispatching
is based on the **apparent type** of the arguments

Overloading

```
public class C {
        public double power(int i, long c){ // do AAA }
        public double power(long x, int c){ // do BBB }
        public double power(long x, long y) { // do CCC }
}       …
```

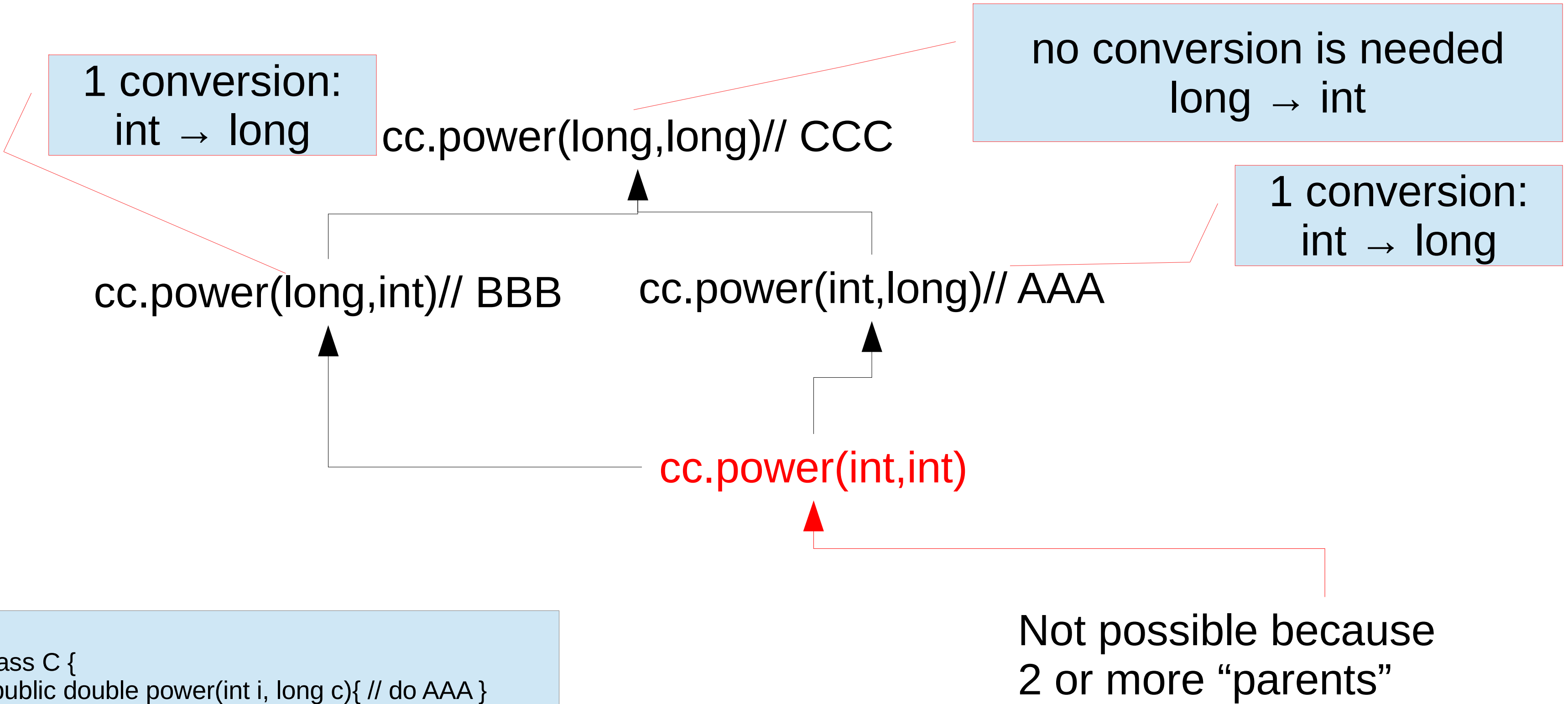
# of conversions	AAA	BBB	CCC
cc.power(long, long)	int → long	int → long	none: OK
cc.power(long,int)	int → long long → int??	none: OK	int → long
cc.power(int, long)	none: OK	int → long long → int??	int → long
cc.power(int, int)	int → long	int → long	int → long int → long

some conversions are impossible

some conversions are the minimum

some conversions are minimal

Most specific method call



```
public class C {
        public double power(int i, long c){ // do AAA }
        public double power(long x, int c){ // do BBB }
        public double power(long x, long y) { // do CCC }
}       …
```

- dispatching is based on number of type conversions

Method dispatching

```
public class C {
        public double power(int i, long c){ // do AAA-1
        public double power(long x, int c){ // do BBB-1
        public double power(long x, long y) { // do CCC-1
}
public class D extends C {
        public double power(int i, long c){ // do AAA-2
        public double power(long x, int c){ // do BBB-2
        public double power(long x, long y) { // do CCC-2
}       …
        C cc = new C();
        D dd = new D();
        C cd = new D();
        int x; long y;  double z;
        cc.power(x, y) // what does it do?
                → AAA-1
        dd.power(x, y)
                → AAA-2
        cd.power(x, y)
                → AAA-2
```

method dispatching
is based on the **apparent type** of the arguments and
real type of instance

Method dispatching: fundamental!

```
ArrayList<Student> theBestOnes = new ArrayList<>();

…

void foo(Collection<Student> awardReceipients){
… // here we assume only Collection, not ArrayList
        ~~awardReceipients.get(0)~~
}
…

foo(theBestOnes);
```

Collection does not have ‘get()’ method

we program using **super types** rather than **concrete types**
to hide un-necessary details