Java

Angelo Gargantini

Informatica III - 2022/2023

M4 java su syllabus

Outline

1 .Language Overview
- History and design goals

2. Classes and Inheritance
- Object features
- Encapsulation
- Inheritance

3. Types and Subtyping
- Primitive and ref types
- Interfaces; arrays
- Exception hierarchy
- Subtype polymorphism and generic programming

- Saltiamo il resto

Origins of the language

- James Gosling and others at Sun, 1990 - 95
- Oak language for “set-top box”
  - small networked device with television display
    - graphics
    - execution of simple programs
    - communication between local program and remote site
    - no “expert programmer” to deal with crash, etc.
- Internet application
  - simple language for writing programs that can be transmitted over network

Design Goals

- Portability
  - Internet-wide distribution: PC, Unix, Mac
- Reliability
  - Avoid program crashes and error messages
- Safety
  - Programmer may be malicious
- Simplicity and familiarity
  - Appeal to average programmer; less complex than C++
- Efficiency
  - Important but secondary

General design decisions

- Simplicity
  - Almost everything is an object
  - All objects on heap, accessed through pointers
  - No functions, no multiple inheritance, no go to, no operator overloading, few automatic coercions
- Portability and network transfer
  - Bytecode interpreter on many platforms
- Reliability and Safety
  - Typed source and typed bytecode language
  - Run-time type and bounds checks
  - Garbage collection

Pro e contro di Java

	Portability	Safety	Simplicity	Efficiency
Interpreted	+	+		-
Type safe	+	+	+/-	+/-
Objects by means of pointers	+		+	-
Garbage collection	+	+	+	-
Concurrency support	+	+		

Java System

- The Java programming language
- Compiler and run-time system
  - Programmer compiles code
  - Compiled code transmitted on network
  - Receiver executes on interpreter (JVM)
  - Safety checks made before/during execution
- Library, including graphics, security, etc.
  - Large library made it easier for projects to adopt Java
  - Interoperability
    - Provision for “native” methods

Java Release History

- 1995 (1.0) – First public release
- 1997 (1.1) – Nested classes
- 2001 (1.4) – Assertions
- 2004 (1.5) – Tiger
  - Generics, foreach, Autoboxing/Unboxing,Typesafe Enums, Varargs, Static Import, Annotations, concurrency utility library
- 2006 (1.6) – Mustang
- 2011 (1.7) – Dolphin

Strings in switch Statement: It enabled using String type in Switch statements

Type Inference for Generic Instance Creation and Diamond Syntax List<Integer> list = new ArrayList<>(); instead of List<Integer> list = new ArrayList<Integer>();

Improvements through Java Community Process

Da java 8

- 2014 (1.8) - Lambda Expressions, collections stream, security libraries, JavaFX
  - Esempio: list.strem().range(1, 4).forEach(System.out::println);
  - **Lamba:** list.forEach((n)->System.out.println(n));
  - Java 9: Modularization
  - Java 10: **Local-Variable Type Inference**
    - **Esempio:**

```
for (var x : arr)
        System.out.println(x + "\n");
```

  - Java 11: **Running Java File with single command**
  - **Java 12: Switch Expressions**

```
switch (x) {
   case 1 -> System.out.println("Foo");
   default -> System.out.println("Bar");
}
```

String quarter = switch (month) { case JANUARY, FEBRUARY, MARCH -> "First Quarter"; //must be a single returning value case APRIL, MAY, JUNE -> "Second Quarter"; case JULY, AUGUST, SEPTEMBER -> "Third Quarter"; case OCTOBER, NOVEMBER, DECEMBER -> "Forth Quarter"; default -> "Unknown Quarter"; };

Da java 12

```
public record Person (String name, String address) {}
```

- Java 13 September 17, 2019
- Java 14 March 17, 2020
- Java 15 September 15, 2020
- Java 16 March 16, 2021
- Java 17 (LTS) September 14, 2021
- Java 18 March 22, 2022
- Java 19 September 20, 2022

```
public String checkShape(Shape shape) {
    return switch (shape) {
        case Triangle t && (t.getNumberOfSides() != 3) -> "This is a weird triangle";
        case Circle c && (c.getNumberOfSides() != 0) -> "This is a weird circle";
        default -> "Just a normal shape";
    };
}
```

Outline

- Objects in Java
  - Classes, encapsulation, inheritance
- Type system
  - Primitive types, interfaces, arrays, exceptions
- Generics (added in Java 1.5)
  - Basics, wildcards, …

Language Terminology

- Class, object -
- Field –
- Method -
- Static members -
- this -
- Package - set of classes in shared namespace
- Native method -

Java Classes and Objects (2)

- Syntax similar to C++
- Object
  - has fields and methods
  - is allocated on heap, not run-time stack
  - accessible through reference (only ptr assignment)
  - garbage collected
- Dynamic lookup
  - Similar in behavior to other languages
  - Static typing => more efficient than Smalltalk
  - Dynamic linking, interfaces => slower than C++

Point Class

```
class Point {
   static public Point O = new Point(0);
   private int x;
   Point(int xval) {x = xval;}      // constructor
   protected void setX (int y)  {x = y;}
   public int  getX()    {return x;}
}
```

- Visibility similar to C++, but not exactly (later slide)

Use of **record** instead of **class**

- As of JDK 14, we can replace our repetitious data classes with records. **Records are immutable data classes that require only the type and name of fields.**
- The *equals*, *hashCode*, and *toString* methods, as well as the *private, final* fields and *public* constructor, are generated by the Java compiler.
- Ex: public record Person (String name, String address) {}
  - will create a class Person with the final fields name and address, the constructor, equals …

Object initialization

- Java guarantees constructor call for each object
  - Memory allocated
  - Constructor called to initialize memory
  - Some interesting issues related to inheritance

We'll discuss later …

- Cannot do this (would be bad C++ style anyway):
  - Obj* obj = (Obj*)malloc(sizeof(Obj));
- Static fields of class initialized at class load time
  - Talk about class loading later

Static fields and methods

- static field is one field for the entire class, instead of one per object.
- static method may be called without using an object of the class
  - static methods may be called before any objects of the class are created. Static methods can access only static fields and other static methods;
- Outside a class, a static member is usually accessed with the class name, as in class_name.static_method(args),

static initialization block

```
class ... {
  /*  static variable with initial value  */
  static int x = initial_value;
  /* --- static initialization block --- */
  static {
/* code to be executed once, when class is loaded */
   }
}
```

- the static initialization block of a class is executed once, when the class is loaded.

Garbage Collection and Finalize

- Objects are garbage collected
  - No explicit *free*
  - Avoids dangling pointers and resulting type errors
- Problem
  - What if object has opened file or holds lock?
- Solution
  - *finalize* method, called by the garbage collector
    - Before space is reclaimed, or when virtual machine exits
    - Space overflow is not really the right condition to trigger finalization when an object holds a lock...)
  - Important convention: call super.finalize
  - Don't design your Java programs such that correctness depends upon "timely" finalization.

Uso di finalize è sconsigliato

- Finalizers are unpredictable, often dangerous, and generally unnecessary.
- Their use can cause erratic behavior, poor performance, and portability problems. Finalizers have a few valid uses, which we'll cover later in this item, but as a rule, you should avoid them. As of Java 9, finalizers have been deprecated, but they are still being used by the Java libraries. The Java 9 replacement for finalizers is cleaners. Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.

Packages and visibility

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier (friendly)	Y	Y	N	N
private	Y	N	N	N

Estensione delle classi (3)

Inheritance

- Similar to Smalltalk, C++
- Subclass inherits from superclass
  - Single inheritance only (but Java has interfaces)
- Some additional features
  - Conventions regarding *super* in constructor and *finalize* methods
  - Final classes and methods

Example subclass

```
class ColorPoint extends Point {
  // Additional fields and methods
   private Color c;
   protected void setC (Color d)  {c = d;}
   public Color  getC()     {return c;}
  // Define constructor
   ColorPoint(int xval, Color cval) {
       super(xval);    // call Point constructor
       c = cval;  }     // initialize ColorPoint field
 }
```

Class *Object*

- Every class extends another class
  - Superclass is *Object* if no other class named
- Methods of class *Object*
  - `getClass` – return the Class object representing class of the object
  - `toString` – returns string representation of object
  - `equals` – default object equality (not ptr equality)
  - `hashCode`
  - clone – makes a duplicate of an object
  - wait, notify, notifyAll – used with concurrency
  - finalize

Importance of hashcode

- Simply put, *hashCode()* returns an integer value, generated by a hashing algorithm.
- Objects that are equal (according to their *equals()*) must return the same hash code. **Different objects do not need to return different hash codes.**
    - If two objects are equal according to equals() method, then their hash code must be same.
    - If two objects are unequal according to equals() method, their hash code are not required to be different. Their hash code value may or may-not be equal.
- If hashcode is not correctly implemented, all the Hash* data structure won’t work.
- Example

Constructors and Super

- Java guarantees constructor call for each object
- This must be preserved by inheritance
  - Subclass constructor must call super constructor
    - If first statement is not call to super, then call super() inserted automatically by compiler
    - If superclass does not have a constructor with no args, then this causes compiler error (yuck)
    - Exception to rule: if one constructor invokes another, then it is responsibility of second constructor to call super, e.g.,

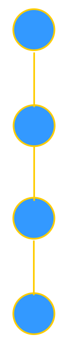
      ```
      ColorPoint() { this(0,blue);}
      ```

      is compiled without inserting call to super
- Different conventions for finalize and super
    - Compiler does not force call to super finalize

Final classes and methods

- Restrict inheritance
  - Final classes and methods cannot be redefined
- Example

  java.lang.String
- Reasons for this feature
  - Important for security
    - Programmer controls behavior of all subclasses
    - Critical because subclasses produce subtypes
  - Compare to C++ virtual/non-virtual
    - Method is "virtual" until it becomes final



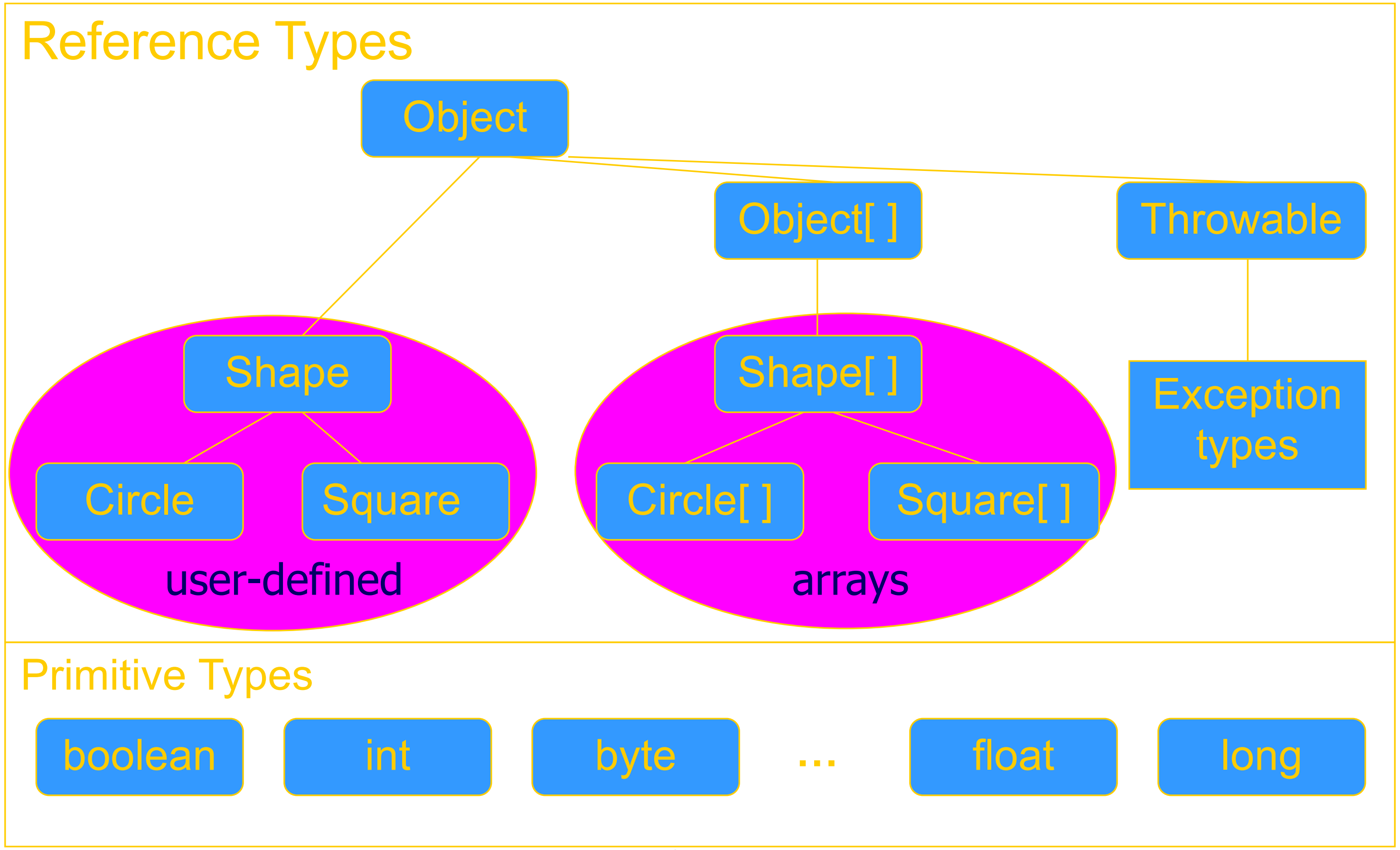
Altri argomenti

- Compatibilità di tipi e conversione
  - Sottoclassi e sottotipi
- Classi astratte e interfacce
- Ereditarietà e ridefinizione dei membri
- Binding dinamico

Java Types

- Two general kinds of times
  - Primitive types – *not* objects
    - Integers, Booleans, etc
  - Reference types
    - Classes, interfaces, arrays
    - No syntax distinguishing Object * from Object
- Static type checking
  - Every expression has type, determined from its parts
  - Some auto conversions, many casts are checked at run time
  - Example, assuming A <: B (A sottotipo di B)
    - Can use A x and type
    - If B x, then can try to cast x to A
    - Downcast checked at run-time, may raise exception

Classification of Java types



Subtyping

- Primitive types
  - Conversions: int -> long, double -> long, ...
- Class subtyping similar to C++
  - Subclass produces subtype
  - Single inheritance => subclasses form tree
- Interfaces
  - Completely abstract classes
    - no implementation
  - Multiple subtyping
    - Interface can have multiple subtypes (extends, implements)
- Arrays
  - Covariant subtyping – not consistent with semantic principles

Java class subtyping

- Signature Conformance
  - Subclass method signatures must conform to those of superclass
- Three ways signature could vary
  - Argument types
  - Return type
  - Exceptions

  How much conformance is needed in principle?
- Java rule
  - Java 1.1: Arguments and returns must have identical types, may remove exceptions
  - Java 1.5: covariant return type specialization

Covariance

- Covariance Definizione
- T si dice covariante (rispetto alla sottotipazione di Java) se ogni volta che A è sottotipo di B allora anche T di A è sottotipo di T B
    - T potrebbe essere il valore ritornato
    - …

Covariance

- Covariance in Java 5
- I valori ritornati da un metodo ridefinito possono essere covarianti
- parameter types have to be exactly the same (invariant) for method overriding, otherwise the method is overloaded with a parallel definition instead.

```
class A {
  public A whoAreYou() {...}
}
class B extends A {
  // override A.whoAreYou *and* narrow the return type.
  public B whoAreYou() {...}
}
```

Array types

- Automatically defined
  - Array type T[ ] exists for each class, interface type T
  - Cannot extended array types (array types are final)
  - Multi-dimensional arrays as arrays of arrays: T[ ] [ ]
- Treated as reference type
  - An array variable is a pointer to an array, can be null
  - Example: Circle[] x = new Circle[array_size]
  - Anonymous array expression: new int[] {1,2,3, ... 10}
- Every array type is a subtype of Object[ ], Object
  - Length of array is not part of its static type

Array subtyping - covariance

- Covariance
  - if S <: T then S[ ] <: T[ ]
    - S <:T means "S is subtype of T"
- Standard type error

```
class A {…}

class B extends A {…}

B[ ] bArray = new B[10]
A[ ] aArray = bArray    // considered OK since B[] <: A[]
aArray[0] = new A()    // compiles, but run-time error
                  // raises ArrayStoreException
// b/c aArray actually refers to an array of B objects
// so that assignment, aArray[0] = new A(); would violate the type of bArray
```

Interfacce (4)

- Java non ammette ereditarietà multipla
- Però posso definere delle interfacce
  - Lista di metodi che definiscono l'interfaccia
  - Ogni interfaccia indentifica un tipo
  - Posso definire sottotipi di interface senza ereditare nulla

Interface subtyping: example

```
interface Shape {
    public float center();
  public void rotate(float degrees);
}
interface Drawable {
    public void setColor(Color c);
  public void draw();
}
class Circle implements Shape, Drawable {
    // does not inherit any implementation
  // but must define Shape, Drawable methods
}
```

Properties of interfaces

- Flexibility
  - Allows subtype graph instead of tree
  - Avoids problems with multiple inheritance of implementations (we will see C++ “diamond”)
- Cost
  - Offset in method lookup table not known at compile
  - Different bytecodes for method lookup
    - one when class is known
    - one when only interface is known
      - search for location of method
      - cache for use next time this call is made (from this line)

Tipi enumerativi (6)

Enumeration

- In prior releases, the standard way to represent an enumerated type was the int Enum pattern

```
// int Enum Pattern - has severe problems!
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

- Not typesafe
- No namespace - You must prefix constants of an int enum with a string (in this case SEASON_)
- Printed values are uninformative

In Java5

public enum Season {
WINTER, SPRING, SUMMER, FALL }

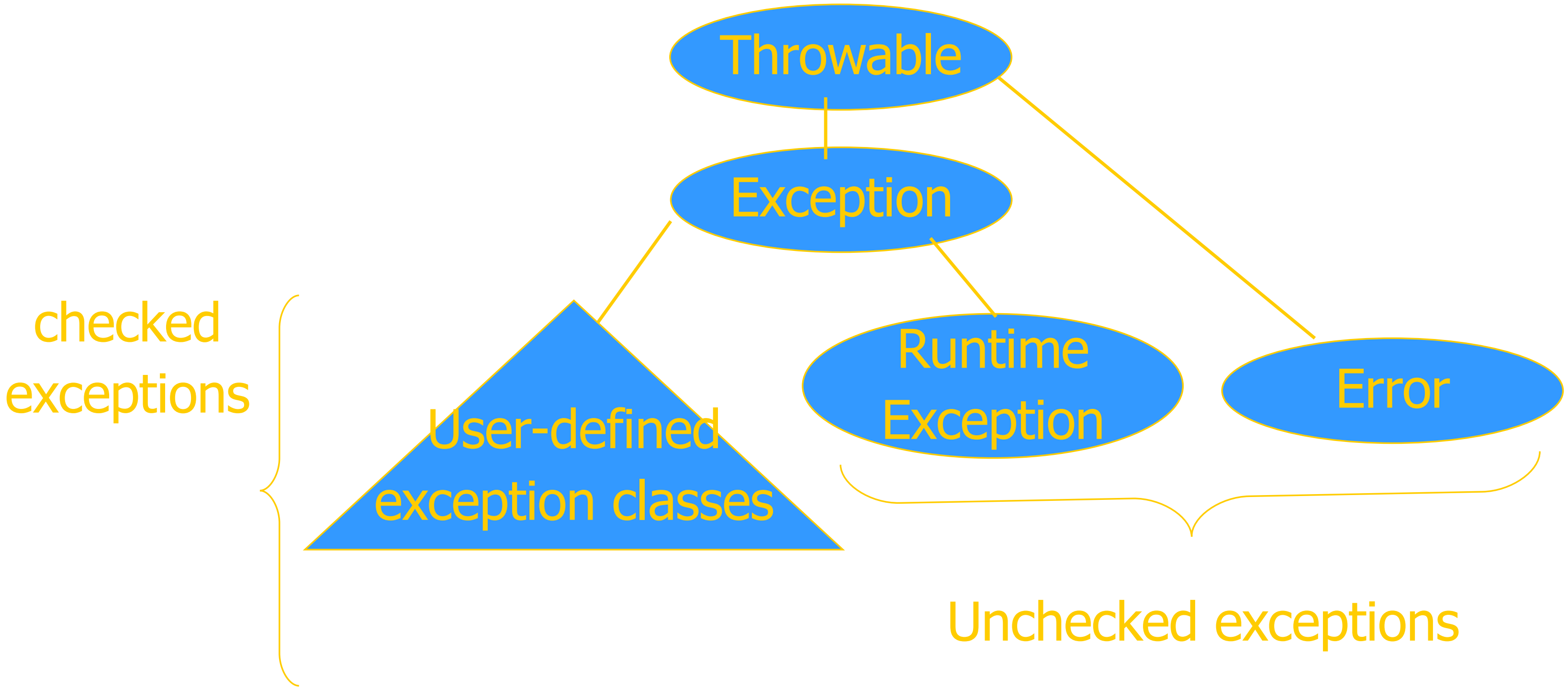
- Comparable
- toString which prints the name of the symbol
- static `values` method that returns an array containing all of the values of the enum type in the order they are declared
  - for (Season s : Season.values()) ...

Eccezioni e asserzioni (12)

Java Exceptions

- Similar basic functionality to ML, C++
  - Constructs to *throw* and *catch* exceptions
  - Dynamic scoping of handler
- Some differences
  - An exception is an object from an exception class
  - Subtyping between exception classes
    - Use subtyping to match type of exception or pass it on …
    - Similar functionality to ML pattern matching in handler
  - Type of method includes exceptions it can throw
    - Actually, only subclasses of Exception (see next slide)

Exception Classes



- If a method may throw a checked exception, then this must be in the type of the method

Try/finally blocks

- Exceptions are caught in try blocks

```
try {
      statements
}     catch (ex-type1 identifier1) {
                  statements
} catch (ex-type2 identifier2) {
                  statements
}   finally {
                  statements
}
```

- Implementation: finally compiled to jsr

Why define new exception types?

- Exception may contain data
  - Class Throwable includes a string field so that cause of exception can be described
  - Pass other data by declaring additional fields or methods
- Subtype hierarchy used to catch exceptions

  catch <exception-type> <identifier> { … }

  will catch any exception from any subtype of exception-type and bind object to identifier

REDEFINIZIONE DEI METODI CON ECCEZIONI

Binding Dinamico in Java

Overload vs Override

- Overlod = più metodi o costruttori con lo stesso nome ma diversa segnatura
  - Segnatura: nome del metodo e lista dei tipi dei suoi argomenti
- L'overloading viene risolto in fase di compilazione
- Esempio

```
public static double valoreAssoluto(double x) {
    if (x > 0) return x;
    else return -x;
}
public static int valoreAssoluto(int x) {
    return (int) valoreAssoluto((double) x);
}
```

Compilazione: scelta segnatura

- In compilazione viene scelta la segnatura del metodo da eseguire in base:

(1) al tipo del riferimento utilizzato per invocare il metodo

(2) al tipo degli argomenti indicati nella chiamata

Esempio

- A r;...
- r.m(2)
- Il compilatore cerca fra tutte le segnature di metodi di nome m disponibili per il tipo A quella "più adatta" per gli argomenti specificati

Esempio

A r;

...

r.m(2)

- Se le segnature disponibili per il tipo A sono:

int m(byte b)

int m(long l)

int m(double d)

- il compilatore sceglie la seconda

- Ricordati che byte << short << int << long<< float<< double

Overriding

- Quando si riscrive in una sottoclasse un metodo della superclasse con la stessa segnatura.
- L'overriding viene risolto in fase di esecuzione
- Compilazione:
- scelta della segnatura: il compilatore stabilisce la segnatura del metodo da eseguire (early binding)
- Esecuzione:
- scelta del metodo: Il metodo da eseguire, tra quelli con la segnatura selezionata, viene scelto al momento dell'esecuzione, sulla base del tipo dell'oggetto (late binding)

Fase di compilazione

(1) Scelta delle segnature “candidate”

- Il compilatore individua le segnature che possono soddisfare la chiamata
  - (a) compatibile con gli argomenti utilizzati nella chiamatail numero dei parametri nella segnatura `e uguale al numero degli argomenti utilizzati ogni argomento `e di un tipo assegnabile al corrispondente parametro
  - (b) accessibile al codice chiamante
- Se non esistono segnature candidate, il compilatore segnala un errore.

(2) Scelta della segnatura “più specifica”

- Tra le segnature candidate, il compilatore seleziona quella che richiede il minor numero di promozioni

Esempio 1

A
assegna(x:long)

B eredita da A
e fa overloading
(stesso nome segnatura diversa)

B
assegna(x:int)
assegna(x:double)

C
assegna(x:int)
assegna(x:double)

C eredita da B
e fa overriding
(stesso nome e segnatura)

A alfa;

- alfa.assegna(2)

Una segnatura candidata:
assegna(long x)

- alfa.assegna(2.0)

Nessuna segnatura candidata (errore)

Esempio 2

A
assegna(x:long)

B
assegna(x:int)
assegna(x:double)

C
assegna(x:int)
assegna(x:double)

B beta;

beta.assegna(2)

Tre segnature candidate:

- assegna(int x)
- assegna(double x)
- assegna(long x)
- La più specifica è assegna(int x)

Ambiguità

- Se per l’invocazione:
- z(1, 2)
- le segnature candidate sono:
- z(double x, int y)
- z(int x, double y)
- Il compilatore non `e in grado di individuare la segnatura pi`u specifica e segnala un messaggio di errore

Esecuzione: scelta del metodo

- La JVM sceglie il metodo da eseguire sulla base della classe dell’oggetto usato nell’invocazione
  - cerca un metodo con la segnatura selezionata in fase di esecuzione
  - risalendo la gerarchia delle classi a partire dalla classe dell’oggetto che deve eseguire il metodo

Esempio 1

A alpha = new B();

alpha.assegna(2l)

EB: segnatura selezionata in A:
assegna(long x)

LB: Ricerca a partire da B un
metodo assegna(long)

Esegue il metodo di A

A
assegna(x:long)

B
assegna(x:int)
assegna(x:double)

C
assegna(x:int)
assegna(x:double)

In questo caso metodo selezionato in EB
ed eseguito coinvidono

Esempio 2

B beta = new C()

beta.assegna(2)

EB: segnatura selezionata
di B: assegna(int x)

LB: Ricerca a partire da C
un metodo assegna(int)

Esegue il metodo di C

Come volevo,
poichè ho ridefinito il metodo

A
assegna(x:long)

B
assegna(x:int)
assegna(x:double)

C
assegna(x:int)
assegna(x:double)

Esempio 3

A alfa = new C()

alfa.assegna(2)

EB Una segnatura candidata: assegna(long x)

LB: Ricerca a partire da C un metodo assegna(long)

Esegue il metodo di A anche se 2 è int !!!

E' dovuto al fatto che non ho ridefinito il metodo di A

A
assegna(x:long)

B
assegna(x:int)
assegna(x:double)

C
assegna(x:int)
assegna(x:double)

Attenzione - Equals

- Quando si ridefiniscono i metodi in java bisogna usare la stessa segnatura !!
- Vedi il problema con equals

```
class A {
 int x;
  A(int y){x = y;}
  public equals(A a){ return (x == a.x);}
}
Object a1 = new A(3);
A a2 = new A(3);
a1.equals(a2);
```

a2.equals (a 1);

Esercizio, corretta implementazione di equals

Outline

- Objects in Java
  - Classes, encapsulation, inheritance
- Type system
  - Primitive types, interfaces, arrays, exceptions
- Generics (added in Java 1.5)
  - Basics, wildcards, …
- Virtual machine
  - Loader, verifier, linker, interpreter
  - Bytecodes for method lookup
- Security issues

Enhancements in JDK 5 (= Java 1.5)

- Enhanced for Loop
  - for iterating over collections and arrays
- Autoboxing/Unboxing
  - automatic conversion between primitive, wrapper types
- Typesafe Enums
  - enumerated types with arbitrary methods and fields
- Varargs
  - puts argument lists into an array; variable-length argument lists
- Static Import
  - avoid qualifying static members with class names
- Annotations (Metadata)
  - enables tools to generate code from annotations (JSR 175)
- Generics
  - polymorphism and compile-time type safety

varargs

- Varargs sono usati per dichiarare un metodo che possa prendere in ingresso un oggetto, n- oggetti o un array di oggetti.
- Esempio
- print(String … s)
- Permette le seguenti chiamate:
- print(“pippo”)
- print(“pippo”,”pluto”)
- print(new String[]{“a”,”b”,”c”})
- Il tipo del parametro formale di un varargs è un array