PROGRAMMAZIONE MULTIPIATTAFORMA

PROGRAMMAZIONE AD OGGETTI

C.D.L. INGEGNERIA E SCIENZE INFORMATICHE

Danilo Pianini — danilo.pianini@unibo.it

Roberto Casadei — roby.casadei@unibo.it

versione stampabile

“Write once, run anywhere...”

Lo slogan, coniato originariamente da Sun Microsystems per illustrare i benefici del linguaggio Java, vale *a patto che*:

- **“write” sia fatta in modo corretto e robusto**;
  - ovvero, sia adottato un approccio di programmazione adeguato.
- **sia possibile distribuire ciascuna applicazione per qualunque piattaforma**.
  - ovvero, sia predisposto un packaging efficace.

Accesso al file system

Path e Separatori

Inserire dei *path assoluti* nel proprio sorgente è **sempre** fonte di problemi quando si scrive software multipiattaforma:

- `C:\Users\UserName\file` — Non funzionerà su piattaforma *nix, e non funzionerà se l'utente Windows non è "UserName".
- `C:\MyProgram\file` — Non funzionerà su piattaforma *nix, e non funzionerà se l'installazione di Windows è sana e il software non è avviato con diritti di amministratore.
- `/home/username/file` — Non funzionerà su piattaforma Windows, e non funzionerà se l'utente non è `username`.

PROBLEMI

- I *separatori* per i path cambiano a seconda dell'OS
- La *struttura del file system* cambia con l'OS
- I *diritti* di lettura e scrittura cambiano con la configurazione

Proprietà di sistema

Java fornisce nella classe `System` un metodo

```
String getProperty(String p)
```

che consente di accedere a proprietà di sistema

Proprietà relative al file system

- `file.separator` — Restituisce `\` per Windows e `/` per Unix
- `java.home` — La directory di installazione di Java
- `user.dir` — La directory da cui il comando `java` è stato invocato
- `user.home` — Restituisce la home directory dell'utente che ha lanciato `java`
- `user.name` — Restituisce il nome utente

```
public static final String PROP_FILE_SEPARATOR = "file.separator";

String separator = System.getProperty(PROP_FILE_SEPARATOR);
```

Accesso ai dettagli del sistema

Funzionalità OS-specifiche

- Talvolta è possibile che in una applicazione si debbano utilizzare librerie non disponibili o non licenziate per alcune piattaforme.
- A supporto di ciò, Java fornisce delle proprietà che consentono di *identificare OS, versione, e JVM corrente*.

PROPRIETÀ RELATIVE AL SISTEMA

- `java.version` — La versione di `java` in uso. Si potrebbe decidere di non usare una funzionalità che si sa non esistere o essere buggata.
- `os.arch` — L'architettura della CPU come rilevata dall'OS (x86, i386, amd64, x86_64, IA64N, ARM, ...)
- `os.name` — Il nome del sistema operativo (Linux, MacOS X, MacOS, Windows 10, Solaris, FreeBSD, ...)
- `os.version` — Restituisce per Windows il numero di versione effettivo (per esempio, Windows 10 restituisce 10.0), per MacOS il numero di versione (per esempio, 10.3.4), per Linux la versione del kernel (es. 6.1)

GUI scalabili e internazionalizzazione

GUI scalabili – Motivazioni

FLESSIBILITÀ

Diversamente dagli anni 90, i dispositivi oggi hanno una *densità di pixel per area* **estremamente** variabile. Si va da 120 *PPI (Pixel Per Inch)* a 640 PPI, su schermi di dimensione estremamente variabile (da 3 a 200 pollici).

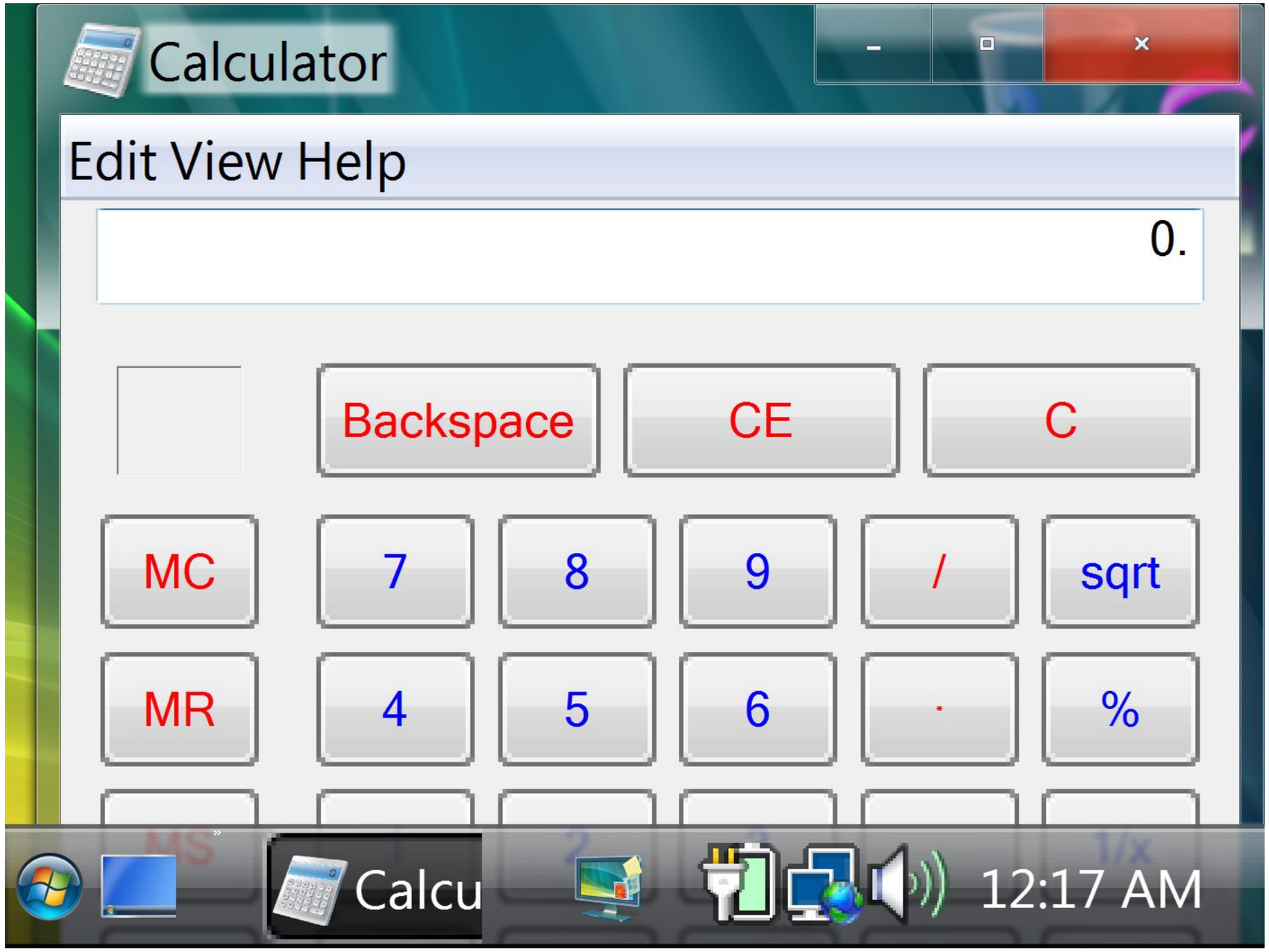
MULTIPIATTAFORMA

Piattaforme diverse, anche a parità di schermo, possono adottare diverse convenzioni:

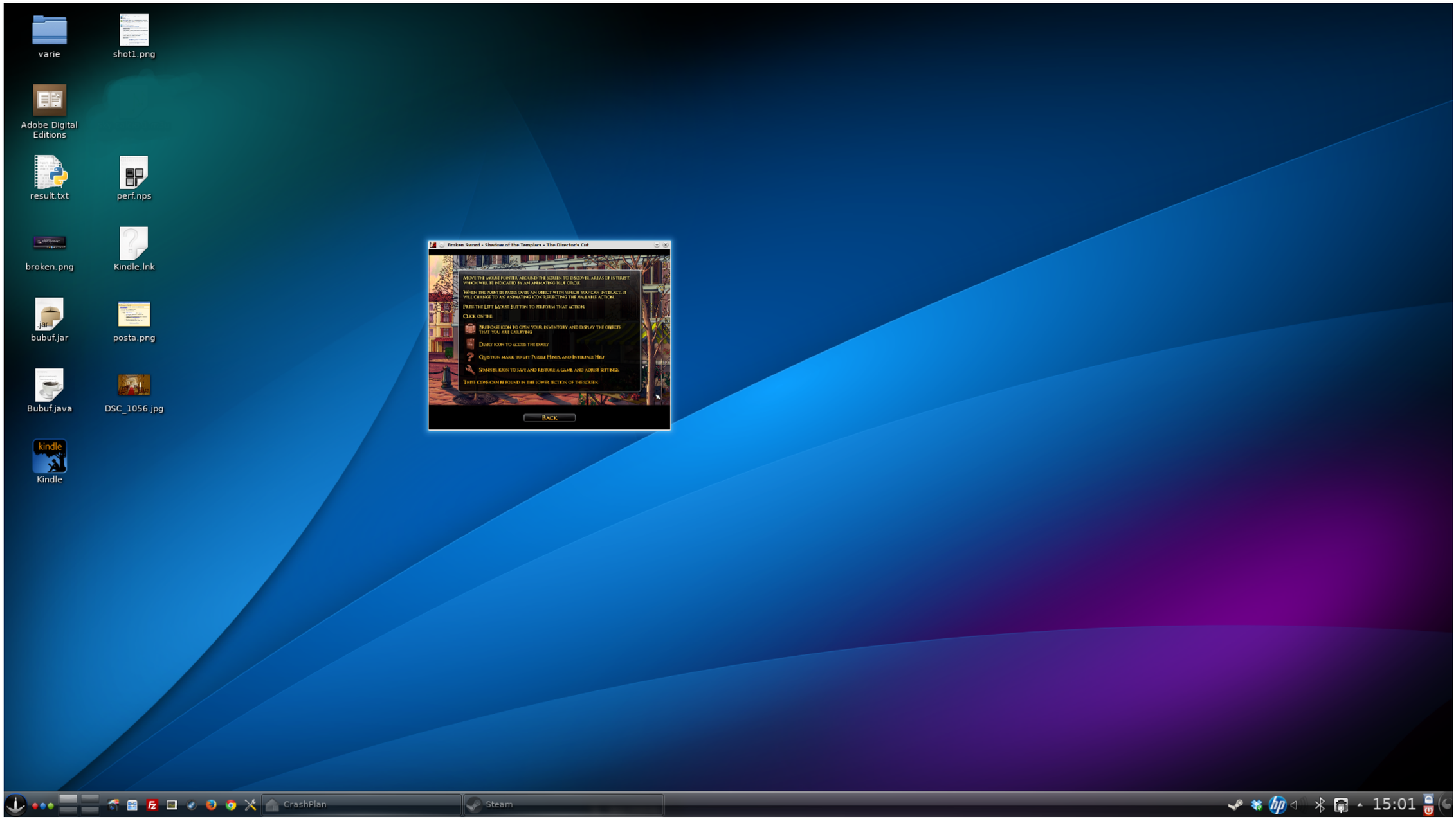
- Diversa grandezza di bordi
- Diversa spaziatura, dimensione e tipo di font
- Diverso sistema di decorazioni

Questi elementi sono stabiliti dal *window manager* (del *windowing system* del sistema utilizzato) e non dallo sviluppatore dell'applicazione. Come indicazione generale vale che **un'applicazione ben sviluppata eredita il "look and feel" dal sistema su cui sta girando**.

Esempio: Finestra non ridimensionabile e bassa risoluzione



Esempio: Finestra non ridimensionabile e alta risoluzione



Best-practices per la costruzione di GUI

- A proposito della specifica delle dimensioni
  - La *dimensione di default* della finestra va calcolata *in base alla dimensione dello schermo*.
  - E' opportuno **non** *specificare dimensioni assolute in pixel* per i componenti della GUI, ma dimensionarli **in termini relativi** *rispetto al container*.
    - Anche per i layout è opportuno non utilizzare dimensioni fisse in pixel.
- A proposito dei font
  - I *font* possono essere allegati all'applicazione
  - La *dimensione dei font* può essere resa *relativa* alla dimensione corrente della view.
- *L'utente deve essere libero di ridimensionare l'interfaccia a piacimento per adattarla alla propria configurazione di schermi*

Supporto multilingua per le UI

- Sarebbe opportuno definire la UI una sola volta e cambiare dinamicamente le parti scritte (il testo) a seconda dell'impostazione della lingua di sistema (o della nostra applicazione).
- In realtà anche per il formato dei numeri, la valuta, le convenzioni sulla data, ...
  - un **locale** è un insieme di parametri che definiscono la lingua dell'utente, la sua regione, e le sue preferenze di visualizzazione delle GUI
  - un esempio di locale è `en_US_UNIX` (lingua inglese, Stati Uniti, piattaforma UNIX)

JAVA RESOURCE BUNDLES

Java fornisce una architettura per l'internazionalizzazione (**i18n = internationalization**), che fa uso di `ResourceBundle` e di una serie di file di supporto (*properties files*).

- per localizzazione (**l10n = localization**) si intende invece il processo di adattare un prodotto internazionalizzato a un particolare locale

Per approfondimenti (per implementare il supporto multilingua):

- https://docs.oracle.com/javase/tutorial/i18n/resbundle/index.html
- http://tutorials.jenkov.com/java-internationalization/resourcebundle.html

CORRETTA CONFIGURAZIONE DI UN PROGETTO GRADLE

Risorse nei progetti Gradle

```
├── src
│   ├── main
│   │   ├── java
│   │   └── resources
│   └── test
│       ├── java
│       └── resources
├── build.gradle.kts
└── settings.gradle.kts
```

- Le cartelle `src/[main|test]/resources` contengono le risorse del progetto opportunamente organizzate
  - Per risorse si intendono icone, file di testo, video, immagini, modelli 3D e qualunque cosa sia necessaria al corretto funzionamento del programma ma non sia una libreria o un file sorgente.

File ancillari per il progetto di OOP

Per OOP, alla struttura del progetto Gradle andranno aggiunti almeno altri due file

```
├── src
│   ├── main
│   │   ├── java
│   │   └── resources
│   └── test
│       ├── java
│       └── resources
├── build.gradle.kts
├── LICENSE
├── README.md
└── settings.gradle.kts
```

README.md

- File con la descrizione del progetto: autori, breve guida d'uso, link a risorse.
  - GitHub è in grado di fare il parse del file e di integrarlo nella pagina del progetto, in modo da dargli una descrizione.

LICENSE

- File con informazioni circa la licenza, necessario affinché il progetto sia open source.
- Per software open source, si raccomanda l'uso di MIT license.
- Qualunque licenza GPL-compatibile è ritenuta idonea per il progetto del corso.

Font ed encoding

Font ed encoding

Le più note piattaforme utilizzano di default encoding diversi:

- **UTF-8** — default su Linux, può essere considerato lo standard de-facto.
- **MacRoman** — default su MacOS, raramente causa artefatti se riconvertito ad altri formati.
- **ISO-8859-1** — default su Windows, può causare artefatti su quasi tutti i caratteri non ASCII se convertito a UTF-8.

ENCODING PER IL CODICE SORGENTE

Solitamente, il codice sorgente si sviluppa utilizzando la codifica UTF-8

- Essenziale se si utilizzano caratteri non inclusi nella tabella ASCII (caratteri accentati, ad esempio).

Configurazione di encoding e newline in VS Code

- `File -> Preferences -> Settings`
- Per l'opzione `Text Editor -> Files -> Encoding` selezionare `UTF-8`
- Per l'opzione `Test Editor -> Files -> Eol` selezionare `LF`
- In basso a destra nella finestra di VS Code c'è l'indicazione della codifica e di EOL per il file selezionato

CARICAMENTO DI RISORSE DAL CLASSPATH

Risorse caricate dal classpath

- Abbiamo visto finora il *classpath* come l'insieme dei *percorsi* dove la virtual machine va a cercare le classi da caricare
  - Come abbiamo visto usando l'opzione `-cp` di `java` e `javac`, il classpath può contenere indifferentemente dei path o dei JAR (o anche degli zip)
- Esso includerà tipicamente anche le risorse del progetto, i JAR delle dipendenze importate, etc.
- Come possiamo accedere a queste risorse in modo uniforme?
  - Ossia caricarle sia che si trovino sul file system, sia che si trovino nel JAR eseguibile, sia che vengano incluse in un JAR di risorse separato.
- Java fornisce un'utilità per caricare risorse dal **classpath**
  - Approccio **location-independent**: non importa dove il codice venga eseguito fin tanto che l'ambiente viene correttamente impostato per trovare le risorse.

`ClassLoader.getSystemResource(AsStream)(String)`

```
public abstract class ClassLoader {
  public static ClassLoader getSystemClassLoader();
  public static URL getSystemResource(String name);
  public static InputStream getSystemResourceAsStream(String name);
  public URL getResource(String name);
  // ...
```

- Un **class loader** (istanza di `ClassLoader`) è un'oggetto responsabile del *caricamento di classi e risorse*
  - ogni class loader ha un class loader padre, per sfruttare un meccanismo di delega
  - il parent di default è il **system class loader** che carica classi e risorse *dal classpath*
- Una **risorsa di sistema** (system resource) è una risorsa "built-in" del sistema software, oppure disponibile nel sistema host (ad es. nel filesystem locale)
  - Per esempio, l'implementazione di base ricerca nel `CLASSPATH`
- L'argomento di `getSystemResource` e `getSystemResourceAsStream` è il **nome di una risorsa** (**non un percorso del filesystem!**), che è una stringa separata da `/` che identifica la risorsa
  - L'interpretazione del nome della risorsa dipende dall'implementazione
  - Il system class loader usa il nome come un path per cercare la risorsa a partire dalle entry del classpath
- `ClassLoader.getSystemResource()` equivale a `ClassLoader.getSystemClassLoader().getResource()`

Risorse caricate dal classpath – Esempi

CARICAMENTO DI FILE

```
final InputStream in = ClassLoader.getSystemResourceAsStream("/settings/settings");
final BufferedReader br = new BufferedReader(new InputStreamReader(in));
final String line = br.readLine();
in.close();
```

CARICAMENTO DI IMMAGINI

```
final URL imgURL = ClassLoader.getSystemResource("/images/gandalf.jpg");
final ImageIcon icon = new ImageIcon(imgURL);
final JLabel lab1 = new JLabel(icon);
```

Progetto di esempio: https://github.com/unibo-oop/example-with-get-resources

Installazione delle impostazioni per-utente

MOTIVAZIONE

Spesso un software ha necessità di caricare al primo avvio delle *impostazioni di default*, quindi lasciare l'utente libero di modificarle e, se avviato successivamente caricare quelle scelte dall'utente. In caso di sistema multiutente, le impostazioni saranno diverse per ciascuno.

STRATEGIA

- Si sceglie una cartella nella **home folder dell'utente** dove salvare le impostazioni.
  - È norma consolidata creare una cartella `.nomeprogramma`.
- Al primo avvio, si verifica se tale cartella esista e se contenga i file di configurazione previsti.
  - Se non è presente, o se non sono presenti e leggibili alcuni i file, si procede a caricare nella cartella di destinazione i file di default dal JAR usando `getResource()`.

DIPENDENZE E LIBRERIE

Dipendenze nel software

> *nos esse quasi nanos gigantium humeris insidentes*
>
> — Bernardo di Chartres

Tutto il software moderno dipende da altro software!

- il sistema operativo
- il *runtime environment* (la Java Virtual Machine)
- le librerie di basa (tutto quello che sta in `java.*` e `javax.*`)
- librerie di terze parti (fra poco)
- risorse esterne (icone, suoni, dati applicativi)

Tutto il software che costruiamo e usiamo dipende da *altro sofware*

- Che dipende da *altro software*
  - Che dipende da *altro software*
    - Che dipende da *altro software*
      - Che dipende da *altro software*
        - Che dipende da *altro software*
          - Che dipende da *altro software*
            - Che dipende da *altro software*
              - Che dipende da *altro software*
                - ...

⇒ Le applicazioni hanno un **albero** di dipendenze!

Un semplice esempio: rate a movie

Proviamo a costruire una *semplice applicazione* che:

1. Richiede come parametro il nome di un film o serie TV
2. Si connette ad Internet e cerca informazioni sul dato prodotto cinematografico
   (per esempio, andando a cercare su IMDb.org o rottentomatoes)
3. Cerca il film che abbiamo fornito come ingresso
4. Stampa tutte le informazioni disponibili su questo film!
   (anno, regista, attori, trama, media voti...)

QUANTO CODICE POTREBBE SERVIRE?

Una possibile soluzione: https://github.com/APICe-at-DISI/sample-gradle-project/blob/master/src/main/java/it/unibo/sampleapp/SimplerRateAMovie.java

Il trucco: usare librerie

- È stata sfruttata una libreria per OMDB
  - Data una chiave, interroga un database di film
  - Nasconde tutta la parte di comunicazione di rete via HTTP, il parsing, eccetera
- Ma a sua volta, questa libreria usa librerie che usano librerie...

```
+--- com.omertron:API-OMDB:1.5
|    +--- commons-codec:commons-codec:1.10
|    +--- org.apache.commons:commons-lang3:3.4
|    +--- com.fasterxml.jackson.core:jackson-core:2.8.7
|    +--- com.fasterxml.jackson.core:jackson-annotations:2.8.7
|    +--- com.fasterxml.jackson.core:jackson-databind:2.8.7
|    |    +--- com.fasterxml.jackson.core:jackson-annotations:2.8.0
|    |    \--- com.fasterxml.jackson.core:jackson-core:2.8.7
|    +--- org.slf4j:slf4j-api:1.7.24
|    \--- org.yamj:api-common:2.1
|         +--- org.apache.httpcomponents:httpclient:4.5.3
|         |    +--- org.apache.httpcomponents:httpcore:4.4.6
|         |    +--- commons-logging:commons-logging:1.2
|         |    \--- commons-codec:commons-codec:1.9
|         \--- org.slf4j:slf4j-api:1.7.24
```

Dipendenze *transitive*

Le dipendenze *indirette* (dipendenze di dipendenze) sono dette *transitive*

In progetti non giocattolo, le dipendenze transitive sono la *maggioranza*

- Si fa molto, molto presto ad avere più di 50 dipendenze

Gestire il classpath diventa molto difficile! Ogni libreria va:

- trovata su Internet
- scaricata
- installata
- aggiunta al classpath di compilazione e di esecuzione

L'applicazione di prima viene lanciata con:

```
java -cp "build/classes/java/main:lib/API-OMDB-1.5.jar:lib/jool-
0.9.14.jar:lib/logback-classic-1.4.1.jar:lib/api-common-2.1.jar:lib/slf4j-api-
2.0.2.jar:lib/httpclient-4.5.3.jar:lib/commons-codec-1.10.jar:lib/commons-lang3-
3.4.jar:lib/jackson-databind-2.8.7.jar:lib/jackson-core-2.8.7.jar:lib/jackson-
annotations-2.8.7.jar:lib/logback-core-1.4.1.jar:lib/httpcore-4.4.6.jar:lib/commons-
logging-1.2.jar" it.unibo.sampleapp.SimplerRateAMovie
```

LA COMPLESSITÀ SFUGGE PRESTO DI MANO!

La gestione delle dipendenze

Ci servirebbe uno strumento capace di:

- *Cercare* (magari in archivi noti) le librerie di cui abbiamo bisogno
- *Scaricare* le suddette (se le trova)
- Costruire il *classpath* e lanciare la *compilazione*
- E magari anche *lanciare l'applicazione*...

Per farlo, però, abbiamo bisogno di conoscere qualche archivio ("*repository*") di librerie, e di sapere come reperirle, ossia conoscere il loro *nome* e *versione*...

Librerie “di terze parti” in Java

Java non definisce alcuno standard per i nomi di libreria

Al compilatore Java e alla JVM (a differenza di quello che accade con altri linguaggi) è *ignoto* il concetto di “libreria”. L’unica astrazione che abbiamo in mano è quella di **classpath**, ma è troppo grezza!

- Usare il solo classpath ci costringerebbe a cercare, scaricare, ed elencare le librerie a mano!

Standard di fatto: Maven naming convention

Quando Java ha preso piede, è stato necessario sopperire a questa mancanza. Un particolare build system, **Apache Maven**, ha elaborato una propria *convenzione* per i nomi, divenuta oggi *sostanzialmente standard* (qualunque build system per Java la adotta).

Maven naming convention

Una libreria Java in formato compatibile con Maven si compone di:

- **groupId**: identifica un (sotto-)progetto o un gruppo di lavoro
  - Segue lo schema di nomi dei package Java, ossia, tipicamente, è un URL invertito
    - ad esempio: `it.unibo`, `com.google`, `io.github`
- **artifactId**: identifica una specifica *libreria* o *modulo di progetto*
  - È un nome semplice in kebab-case
    - ad esempio: `commons-math`, `guava`, `junit-jupiter-assertions-jvm`
- **version**: identifica una specifica *versione* di una libreria
  - possono essere numeri o lettere separati da `.`, `-`, o `+` (solitamente numeri e punti)
    - ad esempio: `1.0`, `1.0.1`, `2.3.5-beta4`, `28ae10dd`, `4.0.2-alpha+28ae10dd`
    - tipicamente (ma non sempre) le versioni con soli numeri e punti sono *stabili*

Per riferirsi ad una libreria specifica, si usa la sintassi: `groupId:artifactId:version`

- `com.google.guava:guava:32-jre`
- `it.unibo.alchemist:alchemist-api:25.0.1`

Ora sappiamo come si chiamano, ma non dove trovarle...

The Maven Central Repository (a.k.a. Sonatype OSSRH)

Assieme alla convenzione per i nomi, Maven definì un *repository* (archivio) dove i creatori di software Java *open source* potessero:

- *pubblicare* le proprie librerie
- *utilizzare* quelle prodotte da sé e da altri ⇐ vedremo come!

La disponibilità e la possibilità di **riuso** ha consentito la nascita dell'"ecosistema" Java, rendendolo uno dei linguaggi/piattaforme di più ampio successo di sempre.

https://central.sonatype.dev/ (storicamente: https://search.maven.org/)

- È possibile trovare moltissime librerie!
- Oltre a scaricarle, sono documentate le *dipendenze*, che andranno a loro volta scaricate...

Sappiamo *dove* trovare le librerie e *come* riferirle, ma ci serve ancora uno strumento per:

- *scaricarle* in automatico
- scaricare anche tutte le *dipendenze transitive*
  - ricorsivamente!
- gestire i classpath che diventano lunghi e complicati

DIPENDENZE E GRADLE

Gestione delle dipendenze in Gradle

Gradle consente di gestire le dipendenze, specificando:

- Dove andarle a reperire (i cosiddetti "*repository*")
- Qual è il loro *nome*
- Qual è la loro *versione*
- Qual è il loro *scopo* (applicazione o testing, compilazione e/o esecuzione)
  - Infatti, potremmo avere delle librerie che ci servono solo per dei test, ma di cui possiamo fare a meno una volta che l'applicazione è testata
    - I produttori di automobili provano il motore sui banchi prova, ma quando comprate la macchina non ve lo danno: col software è analogo...

Specificare i repository in Gradle

In Gradle è possibile “puntare” ad archivi di librerie specificandolo in un blocco `repositories`

- Maven Central è un membro speciale, e la sua configurazione è semplificata.

Per dire a Gradle di:

1. Preparare il necessario per gestire un progetto Java
2. Configurare Maven Central come *repository* per le eventuali liberie

è sufficiente configurare `build.gradle.kts` come segue:

```
plugins { java } // Carica il necessario per Java
repositories { mavenCentral() } // Configura Gradle per cercare e scaricare da Maven Central
```

Dipendenze in Gradle: preparazione

Siamo pronti per importare le librerie che vogliamo! Dobbiamo solo:

1. Trovare su Maven Central la libreria che intendiamo usare e annotare `groupId`, `artifactId`, e `version`
   - per questo esempio: la libreria OMDB di "Omertron"
     - https://central.sonatype.dev/artifact/com.omertron/API-OMDB/1.5
     - `com.omertron:API-OMDB:1.5`
2. Capire quale sia il suo *scope*

Scopo delle dipendenze in Gradle

Gradle consente di (costringe a) dire chiaramente "a cosa serve" una certa libreria. Noi vedremo solo alcuni degli scope disponibili:

- `implementation`: la libreria ci serve sia per compilare che per eseguire la nostra applicazione
  - è la scelta *più comune*
  - in questa lezione useremo solo questo scope
- `testImplementation`: la libreria ci serve per compilare ed eseguire i test
  - ma non deve essere usabile quando si scrive l'applicazione vera e propria
    - Sarebbe come costruire un motore che richiede pezzi del banco prova per funzionare…
- `testRuntimeOnly`: la libreria ci serve per eseguire i test (sarà nel `-cp` di `java`), ma non per compilarli (*non* sarà nel `-cp` di `javac`)

Dipendenze in Gradle: specifica

Una volta identificata la libreria

- `com.omertron:API-OMDB:1.5`

e scelto lo scope che vogliamo usare

- `implementation`

Possiamo semplicemente configurare Gradle per importarla dentro il blocco `dependencies`:

```
plugins { java } // Carica il necessario per Java
repositories { mavenCentral() } // Configura Gradle per cercare e scaricare da Maven Central
dependencies {
  implementation("com.omertron:API-OMDB:1.5")
}
```

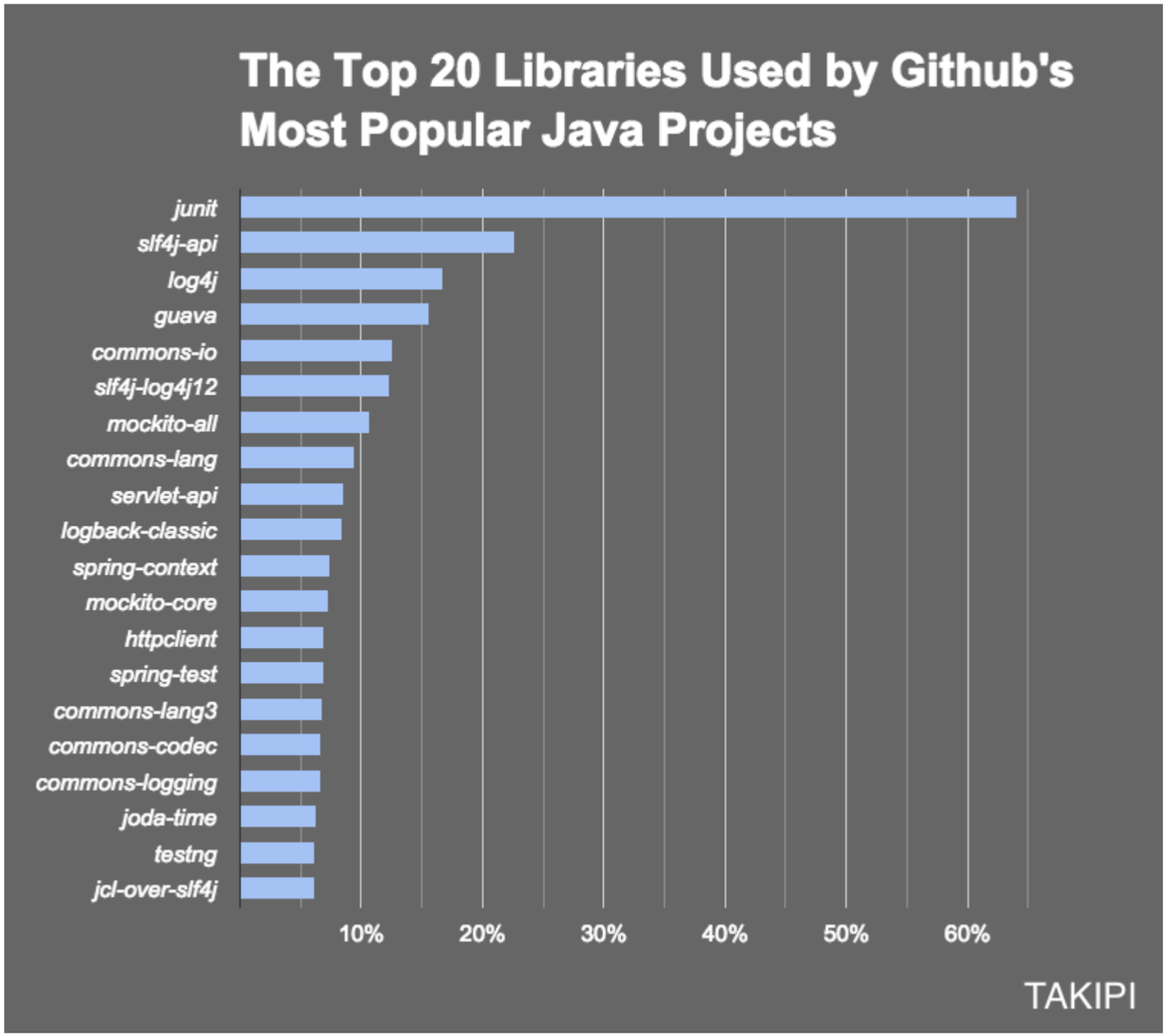
Quando lanceremo il task `compileJava`, Gradle si occuperà di:

- scaricare la libreria
- salvarla nella cartella dell'utente per uso futuro
- includerla nel classpath di compilazione!

Librerie particolarmente utili

- **Google Guava** (https://github.com/google/guava)
  - Il progetto Guava di Google raccoglie diverse librerie core sviluppate da Google che possono essere utilizzate per lo sviluppo di applicazioni.
  - Ad esempio, sono disponibili librerie per Collections Management, Concurrency, I/O, String Processing, ...
- **Apache Commons** (https://commons.apache.org)
  - Estensioni al linguaggio (Commons Lang3)
  - Libreria matematica estesa (Commons Math3)
  - Accesso semplificato all'I/O (Commons IO)
  - Costruzione semi automatica di una command line (Commons CLI)
  - Encoding e crittazione (Commons Codec), compressione (Commons Compress)
- **Static Logger Facade for Java (SLF4J)** (http://www.slf4j.org)
  - Backend-independent logging (addio `println`)

Top 20 Java Library



Awesome Java

Esiste una lista, costantemente manutenuta, che elenca le più comuni, diffuse e stabili librerie per una pletora di usi: https://bit.ly/awesome-java

USATELE!

- **Usare librerie e non reinventare la ruota è IMPORTANTE** e valutato positivamente.
- Attenzione però a scegliere le librerie dopo aver fatto il **modello del dominio** dell'applicazione: **PRIMA** si studia il problema, **DOPO** si implementa una soluzione: siete aspiranti ingegneri, cercate di lavorare sempre top-down quando possibile, non partite dalla libreria per costruirci sopra un software, ma partite dai requisiti e - se utile - sfruttate le librerie per soddisfarli.

ATTENZIONE AI FRAMEWORK!

Alcune librerie sono costruite come *Framework*, ossia come ossature di applicazioni, pensate per velocizzare la costruzione di un certo tipo di software

- Esempio tipico: l'engine per videogames libGDX

Uno degli scopi del progetto di OOP è quello di misurare se siate bravi designer, ma per farlo è necessario che il design della vostra applicazione l'abbiate fatto voi e non chi ha costruito il framework.

Vi raccomandiamo quindi di **evitare i framework**! O, al più, usarli solo *dopo che il progetto è avviato* come semplice libreria (non semplice e non sempre possibile)