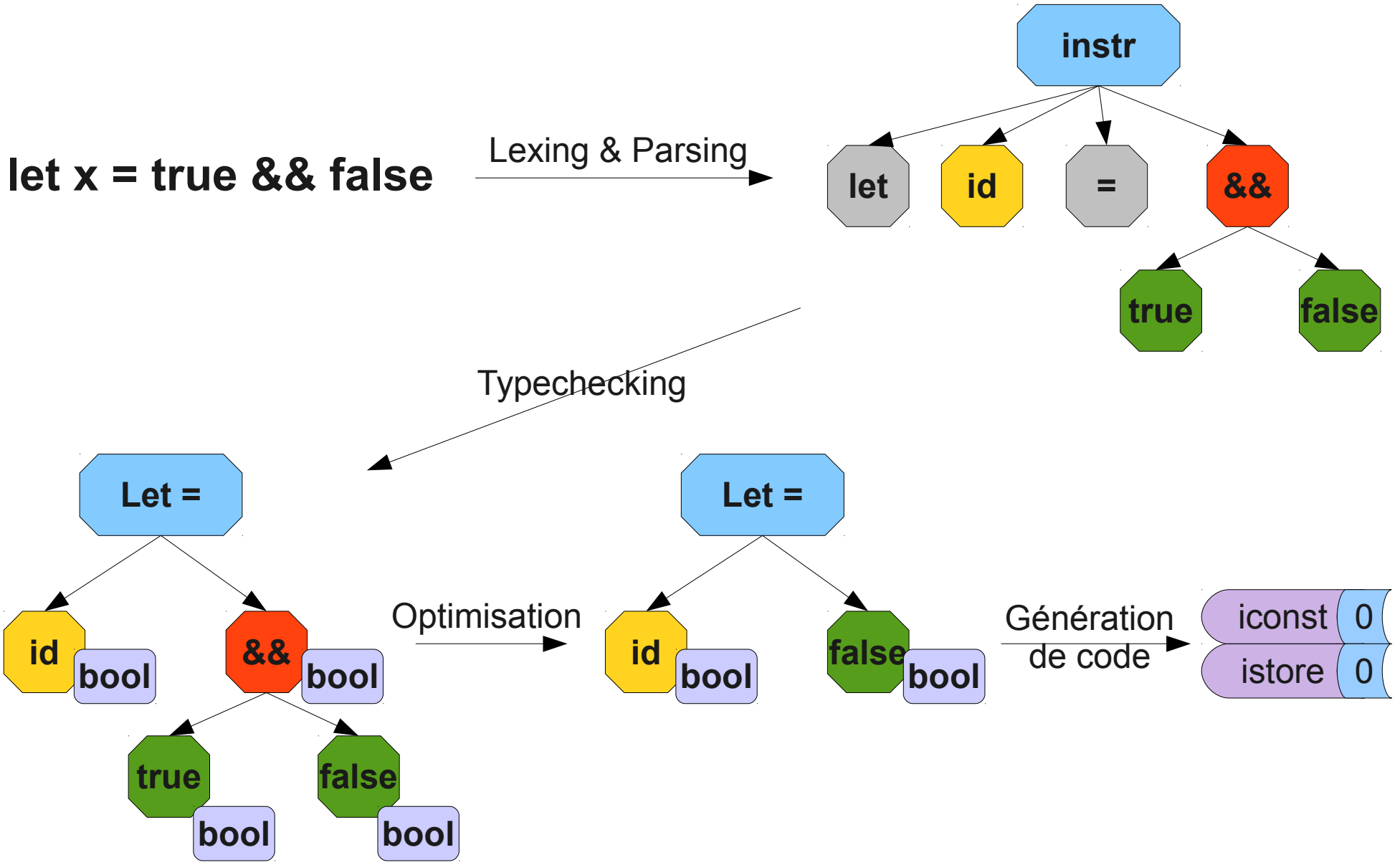
Génération de code

Rémi Forax
Matthieu Constant

Plan

- Code intermédiaire
- Bytecode/Jasmin
- Variables locales
- Structures de contrôle
- Appels de méthodes

Rappel


let x = true && false
Lexing & Parsing
instr
let
id
=
&&
true
false
Typechecking
Let =
id
bool
&&
bool
true
bool
false
bool
Optimisation
Let =
id
bool
false
bool
Génération
de code
iconst
0
istore
0

Code intermédiaire

- Code proche de l'assembleur des machines cibles mais pas spécifique à une machine
- Il existe plusieurs sortes de codes intermédiaires

- Caractéristiques communes :
  - Nombre illimité de variables (registres)
  - Instructions de sauts (conditionnels ou non) pour représenter les tests et les boucles

Codes intermédiaires

- Plusieurs sortes de code intermédiaire
  - Code 3 adresses
    - représentation à base de registres
  - Code à base d'arbre,
    - Représentation avec une pile
  - Static Single Assignment form (SSA)
    - 1 variable ne peut être affecté qu'une seul fois
- Dans un vrai compilateur, plusieurs formes peuvent co-exister

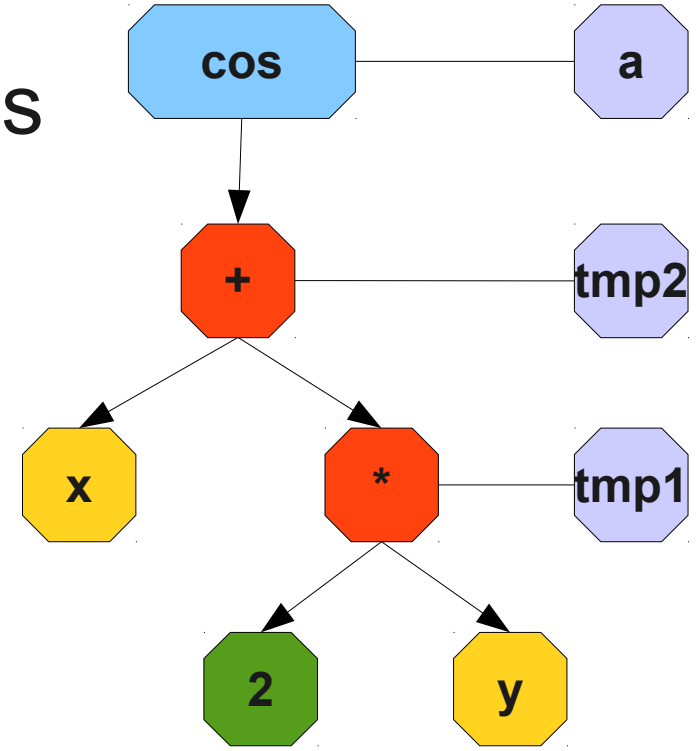
Code 3 adresses

- On doit stocker les résultats intermédiaires des expressions dans une variable temporaire

  a = cos(x+2*y)

  devient

  tmp1 = 2*y
  tmp2 = x + tmp1
  a = cos(tmp2)



- Il faut une variable temporaire par nœud intermédiaire de l'AST

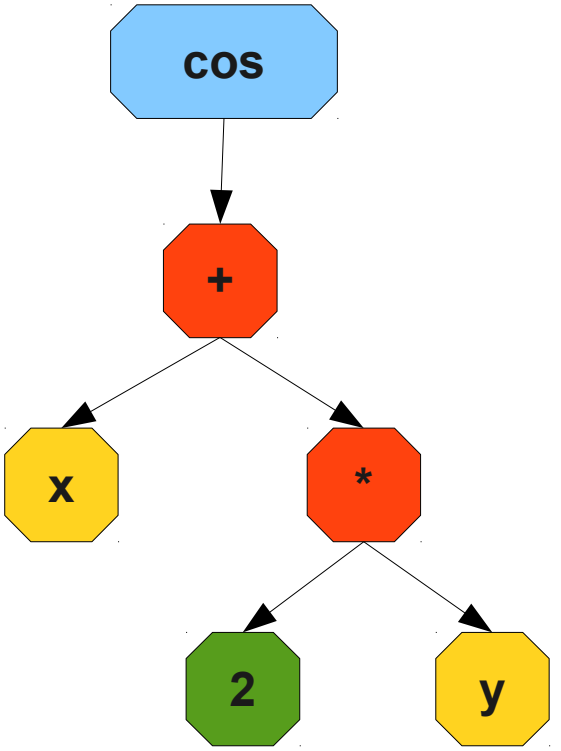
Code à base d'arbre

- Les résultats intermédiaires sont stockés sur la pile

  a = cos(x+2*y)

  devient

  iload x
  iload y
  iconst_2
  imult
  iadd
  invokestatic Math.cos (I)I



Static Single Assignment

- Permet d'exprimer facilement de nombreuses optimisations

  y = 1
  y = 2
  x = y

  est transformé en

  y1 = 1
  y2 = 2
  x1 = y2

- Si seul x est utilisé, y1 = 1 peut être supprimé

Structure de contrôle

- Les structures de contrôle sont remplacées par du code basé sur des tests et des *goto*

- Le if

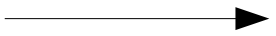
```
int a;
if (test) {
  a = 3;
} else {
  a = 4;
}
print a;
```

```
if (test) goto test

a = 4;
goto end

test:
a = 3;

end:
print a
```

Structure de contrôle

- Boucle for:

```
for (int i = 0;i < 5 ;i = i + 1) {
  print i;
}
print "end";
```



```
i = 0

test:
if (i >= 5) goto end

print i;

continue:
i++;

goto test:

end:
print "end";
```

Bytecode

- Flot d'instructions binaires représentant un programme
- Exécutable par une JVM (Java Virtual Machine)
- Code intermédiaire utilisant une pile de cadres (stack frames)
- Un cadre est associé à un appel de méthode
  - Variables locales
  - Pile d'opérandes

Anatomie d'un .class

- Un dictionnaire de constantes utilisées dans les instructions de la classe (constant pool)

- Une description de la classe
  - Modificateurs, nom, héritage etc,
  - Description des champs
  - Description des méthodes

Exemple de bytecode

- Exemple de HelloWorld avec javap

- public HelloWorld();
  Code:
  0: aload_0
  1: invokespecial #21; //Method **java/lang/Object."<init>":()V**
  4: return

- public static void main(java.lang.String[]);
  Code:
  0: getstatic #14; //Field **java/lang/System.out:Ljava/io/PrintStream;**
  3: ldc #20; //String HelloWorld
  5: invokevirtual #23; //Method **java/io/PrintStream.println:(Ljava/lang/String;)V**
  8: return

Le constant pool de l'exemple

- Le constant pool est un tableau ou chaque ligne contient un type d'items
  - Class, Fieldref, Methodref, InterfaceMethodref, String, Integer, Float, Long, Double, NameAndType, Utf8
  - Un item peut lui-même référencé d'autres items

```
const #1 = NameAndType #24:#26;//  out:Ljava/io/PrintStream;
const #2 = Asciz   ([Ljava/lang/String;)V;
const #3 = Asciz   java/lang/Object;
const #4 = Asciz   <init>;
const #5 = class   #3;  //  java/lang/Object
const #6 = NameAndType #4:#9;//  "<init>":()V
const #7 = class   #18;//  java/io/PrintStream
const #8 = Asciz   Helloworld.j;
const #9 = Asciz   ()V;
...
const #21 = Method    #5.#6;    //  java/lang/Object."<init>":()V
```

Outils

- Assembleur Java
  - Jasmin
- APIs Java de génération de bytecode
  - ASM, BCEL, SERP
- Afficheur de bytecode Java
  - javap
  - ASM bytecode outline (plugin eclipse)

Jasmin

```
.class public HelloWorld
.super java/lang/Object
```

- ```
  ; ceci est un commentaire
  .method public <init>()V
     aload_0
     invokespecial java/lang/Object/<init>()V
     return
  .end method
  ```

- ```
  .method public static main([Ljava/lang/String;)V
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "HelloWorld"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    return
  .limit stack 2
  .limit locals 1
  .end method
  ```

Format des descripteurs

- Nom d'une classe
  java/lang/Object
- Descripteur de champs
  B byte, **C** char
  D double, **F** float
  I integer, **J** long
  S short, **Z** boolean

  Ljava/lang/Object; classe java.lang.Object
  [I tableau d'entiers

Format des descripteurs

- Descripteur de method
  (*descs*)*desc_ou_V*
- Exemple:
  ()V renvoie void
  (I)Z prend un int et renvoie un booléen
  (II)V prend deux ints
  (Ljava/lang/Object;I)V prend Object et un int

Opérations sur la pile

- Toutes les opérations s'effectuent sur la pile
- Chaque case de la pile fait 32 bits
  - Les boolean, byte, short, int, float, référence à objet prennent 1 case
  - les long et double prennent 2 cases

- Exemple:

```
iconst_1      // empile 1
iconst_2      // empile 2
iadd          // dépile 1 et 2 et empile 3 (=1+2)
```

Les constantes

- charger une constante su sommet de la pile :
  - iconst_0, iconst_1 ... iconst_5, iconst_m1
  - bipush [-128, +127]
  - sipush [-32728, +32727]
  - Ldc, pour toutes les constantes contenues dans une case
  - ldc2_w, pour toutes les constantes contenues dans deux cases
- Les booléens sont gérés comme des ints

Variables locales

- Chaque cadre (stack frame) a un tableau de registres (contenant paramètres/variables locales)
- Le contenu des variables est acessible via son indice dans le tableau (0, 1, ...)
- Typage
  - Les boolean/char/byte/int/référence à objet prennent 1 case
  - Les doubles et les longs prennent 2 cases

Variables locales

- Deux opérations possibles
  - Chargement (load n) : lit la variable à l'indice n et l'empile
  - Sauvegarde (store n) : dépile une valeur et la stocke dans une variable à l'indice n

- Les opérations sont typées à l'aide d'un préfixe

  iload/istore -> int, lload/lstore -> long,
  dload/dstore -> double, aload/astore -> objet

Exemple

```
.method public static main([Ljava/lang/String;)V
iconst_2                    ;empile 2 (1 case)
istore 1                        ;case 1: 2 (int)
ldc2_w 2.0                  ;empile 2.0 (2 cases)
dstore 1                    ;cases 1 et 2: 2.0 (double)
….
return
.end method
```

Vérificateur

- Pour qu'une classe soit valide pour la machine virtuelle, il faut
  - Que l'on applique les opérations sur les bons types dans la pile et sur les variables locales
  - Que l'on indique le nombre maximal de variables locales et la taille maximale de la pile
- Dans le cas contraire, le vérificateur de bytecode de la machine virtuelle ne permettra pas le chargement de la classe

Limites (stack/locals)

- Pour chaque méthode, on a besoin d'indiquer :
  - Le nombre maximum de variables locales (locals)
  - La taille maximale de la pile (stack)

```
.method public static main([Ljava/lang/String;)V
iconst_2        ;empile 2 (1 case)
istore 1      ;case 1: 2 (int)
ldc2_w 2.0      ;empile 2.0 (2 cases)
dstore 1        ;cases 1 et 2: 2.0 (double)
return
.limit stack 2
.limit locals 3
.end method
```

Manipulation de la pile

- Il existe des instructions pour dupliquer le sommet de pile
- dup
  - duplique 1 valeur
- dup2
  - duplique 2 valeurs ou 1 seul long/double

Opérations

- Les opérations numériques sont typées (préfixes i,l,f,d)
  - **i**add (addition), **i**sub (soustraction), **i**mul (multiplication), **i**div (division entière), **i**rem (reste)
- Opération sur les ints
  - iand (&), ior (|), ixor(^), ineg(~), ishl(<<), ishr(>>), iushr(>>>)
- i++, ou i+=2 existe !
  - iinc 3 2 (ajoute 2 à la variable locale 3)

Conversions

- Comme les opérations et les variables sont typés, il faut parfois faire des conversions

- i2b, i2c, i2s, i2d, i2f, i2l,
  l2i, l2f, l2d,
  f2i, f2l, f2d,
  d2i, d2l, d2f

- On ne peut faire les conversions vers byte/char/short qu'à partir d'un int

Branchements

- Inconditionnels
  - goto <label>: on va directement au label <label> dans le code
- Conditionnels
  - ifeq, ifne, iflt, ifle, ifgt, ifge (*dépile et compare à 0*)
  - if_acmp[eq|ne] (*dépile 2 objets et compare les références*)
  - if_icmp[eq|ne|lt|le|gt|ge] (dépile deux entiers et les compare)

Exemple

```
iconst_2
istore 1
goto end
ldc2_w 2.0
dstore 1
end:
iload 1
iconst_1
iadd
….
```

Exemple

```
iconst_2
istore 1
iconst_0
ifeq end
ldc2_w 2.0
dstore 1
end:
iconst_2
iconst_1
iadd
```

Comparaison des long/double

- Le test se fait en deux fois :
  - On compare deux valeurs avec lcmpg/dcmpg, qui dépile deux valeurs, les compare et empile un entier (-1,0 ou +1)
  - puis on teste l'entier avec ifeq/ifne...

```
ldc2_w 2.0
dstore_2
dload_2
ldc2_w  4.0
dcmpg
ifgt label1
…
label1:
….
```

Appels de méthodes

- Création d'un nouveau cadre (à l'appel)
  - Les valeurs des paramètres sont dépilées du cadre appelant et placées dans les variables locales du nouveau cadre
  - Le nouveau cadre devient courant
- Destruction du cadre (à la fin de la méthode)
  - La valeur de retour de la méthode est empilée dans le cadre appelant
  - Le cadre appelant redevient courant

Appels de méthodes

- Il existe différents types d'appels de méthodes
  - *invokespecial* (constructeur, méthodes de la super-classe et méthodes privées de la classe courante)
  - *invokestatic* (méthodes statiques)
  - *invokevirtual* (méthodes « normales »)
  - etc.

- Les arguments sont stockés dans la pile
- Pour *invokespecial* et *invokevirtual*, l'objet contenant la méthode se trouve dans la pile

Exemple

```
ldc "abcdefg"
iconst_2
iconst_3
invokestatic java/lang/Math/max(II)I
invokevirtual java/lang/String/charAt(I)C
```

Allocation d'un objet

- *new Integer(2)* est séparé en deux opérations
  - new
  - invokespecial <init> pour appeler le constructeur

- Exemple :

```
new java/lang/Integer
dup
iconst_2
invokespecial java/lang/Integer/<init>(I)V
astore 1
```

Checkcast et instanceof

- Checkcast permet de faire un cast
  - Exemple

```
aload_1
checkcast     java/lang/Integer
```

- Instanceof fait le test instanceof
  - Exemple

```
aload_1
instanceof  java/lang/Integer
```

Création de tableaux

- Tableau de types primitifs
  - newarray <type> (type=boolean,int,float, etc.)

    iconst_5
    newarray int ; on alloue un tableau de 5 ints

- Tableau d'objets
  - anewarray <classe>

    iconst_5
    anewarray java/langString ; on alloue un tableau de 5 String

Accès aux valeurs des tableaux

- baload (byte et boolean), caload, iaload, lalod, faload, daload, aaload
- bastore (byte et boolean), castore, iastore, lastore, fastore, dastore, aastore
- Exemple:

```
iconst_5
newarray int
astore_1
aload_1
iconst_3  ; indice de la case
iconst_5
iastore
aload_1
iconst_3   ; indice de la case
iaload
```

Accès aux champs

- getfield/putfield pour les champs d'instance
- getstatic/putstatic pour les champs static

- Exemple:

```
getstatic java/lang/System/out  Ljava/io/PrintStream;
ldc "HelloWorld"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

switch

- Il existe deux implantations des switchs
  - tableswitch (table de branchement)
  - lookup switch (dichotomie sur les branchements)

- Si les valeurs sont continues (0,1,2,3,...)

  => tableswitch

lookupswitch

```
lookupswitch
    3 : label1
    7 : label2
    default : defLabel

label1:
    ; case 3
    goto break

label2:
    ; case 7
    goto break;

defLabel:
    ; default

break:
```

tableswitch

```
tableswitch 0
    label1
    label2
    default : defLabel

label1:
    ; case 0
    goto break

label2:
    ; case 1
    goto break;

defLabel:
    ; default

break:
```

Ecrire ses propres méthodes

- Les paramètres sont considérés comme des variables locales (premier paramètre → indice 0 si méthode statique, 1 sinon)
- Si méthode non-statique, la référence à *this* est dans la variable locale 0

- Pour retourner une valeur, on utilise
  - ireturn, dretur, areturn, etc.
- Si la méthode ne retourne pas de valeur, on utilise *return*

Exemple

```
.method private static add5(I)I
iload_0
iconst_5
iadd
ireturn
.limit locals 1
.limit stack 2
.end method
```

Exemple (suite)

```
.method public static main([Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
bipush 10
invokestatic HelloWorld/add5(I)I
invokevirtual java/io/PrintStream/println(I)V
return
.limit stack 3
.limit locals 3
.end method
```

Revenons à HelloWorld

```
.class public HelloWorld
.super java/lang/Object

; ceci est un commentaire
  .method public <init>()V
     aload_0
     invokespecial java/lang/Object/<init>()V
     return
  .end method

.method public static main([Ljava/lang/String;)V
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ldc "HelloWorld"
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  return
.limit stack 2
.limit locals 1
.end method
```