

Programación orientada a objetos: Java

Paradigmas de la Programación

FaMAF-UNC 2021

capítulo 13

basado en filmas de John Mitchell

orígenes

- James Gosling y otros en Sun, 1990 - 95
- lenguaje Oak para la "set-top box"
 - dispositivo chico en red con pantalla televisor
 - gráficos
 - programas simples
 - comunicación entre el programa local y un sitio remoto
 - programadores no expertos
- aplicación a internet
 - lenguaje sencillo para escribir programas que se pueden enviar por la red

objetivos de diseño

- portabilidad
 - a todo el internet: PC, Unix, Mac
- confiabilidad
 - evitar crashes y mensajes de error
- seguridad
 - programadores maliciosos
- simplicidad y familiaridad
 - atractivo para programadores, más sencillo que C++
- eficiencia
 - importante pero secundaria

decisiones de diseño generales

- simplicidad
 - casi todo es un objeto
 - los objetos están en el heap, y se acceden a través de punteros
 - no hay funciones, ni herencia múltiple, ni go to, ni sobrecarga de operadores y pocas coerciones automáticas de tipo
- portabilidad
 - el intérprete de bytecode está en muchas plataformas
- confiabilidad y seguridad
 - código fuente tipado y lenguaje bytecode tipado
 - tipado en ejecución
 - recolección de basura

el sistema Java

- lenguaje de programación Java
- compilador y sistema de ejecución
 - el programador compila el código
 - el código compilado se transmite por la red
 - el receptor lo ejecuta en el intérprete (JVM)
 - comprobaciones de seguridad antes y durante la ejecución
- biblioteca, incluyendo gráficos, seguridad, etc.
 - una biblioteca extensa que hace fácil adoptar Java para proyectos
 - interoperabilidad

contenidos

- objetos en Java
 - Clases, encapsulación, herencia
- sistema de tipos
 - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
 - básicos, wildcards, ...
- máquina virtual
 - Loader, verifier, linker, interpreter
 - Bytecodes para lookup de métodos
- temas de seguridad

contenidos

objetos en Java

- Clases, encapsulación, herencia
- sistema de tipos
 - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
 - básicos, wildcards, ...
- máquina virtual
 - Loader, verifier, linker, interpreter
 - Bytecodes para lookup de métodos
- temas de seguridad

terminología

- clase, objeto - como en los otros lenguajes
- campo – miembro datos
- método – miembro función
- miembros estáticos – campos y métodos de la clase
- this - self
- paquete – conjunto de clases en un mismo espacio de nombres (namespace)
- método nativo – método escrito en otro lenguaje

objetos y clases

- sintaxis semejante a C++
- objeto
 - tiene campos y métodos
 - alojado en el heap, no en la pila de ejecución
 - se accede a través de referencia (es la única asignación de puntero)
 - con recolección de basura
- lookup dinámico
 - comportamiento semejante a otros lenguajes
 - tipado estático => más eficiente que Smalltalk
 - linkeado dinámico, interfaces => más lento que C++

clase Punto

```
class Point {  
    private int x;  
    protected void setX (int y)    {x = y;}  
    public int  getX()              {return x;}  
    Point(int xval) {x = xval;}    // construct  
};
```

inicialización de objetos

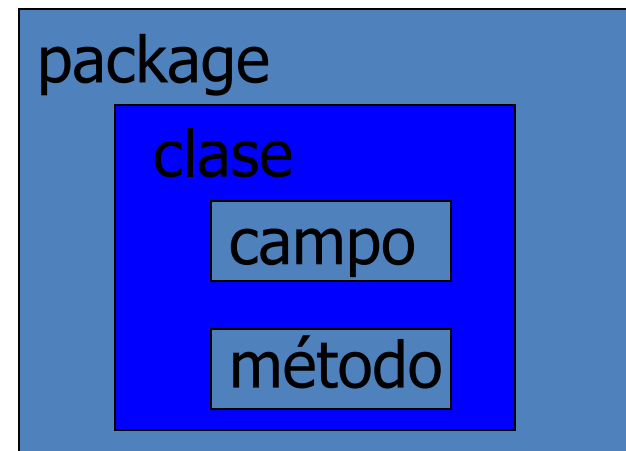
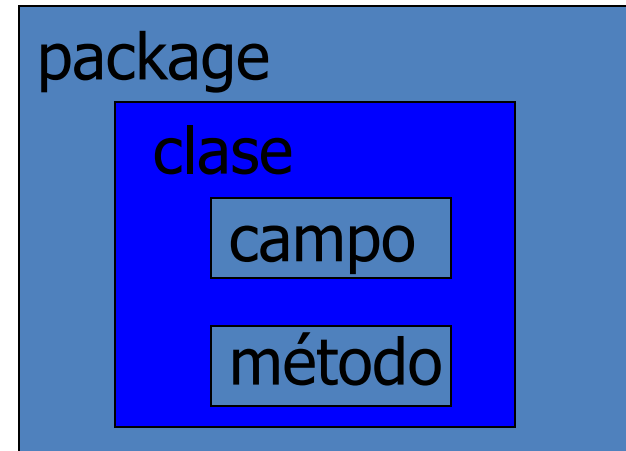
- Java garantiza la llamada al constructor para cada objeto
 - se aloja memoria
 - se llama al constructor para inicializar memoria
- los campos estáticos de la clase se inicializan en tiempo de carga

Garbage Collection y *Finalize*


- los objetos pasan por la recolección de basura
 - no hay una instrucción *free* explícita
 - evita punteros colgantes
- problema
 - qué pasa si un objeto ha abierto un archivo o tiene un lock?
- solución
 - método *finalize*, llamado por el garbage collector

encapsulación y packages

- todos los campos y métodos pertenecen a una clase
- cada clase es parte de algún package
 - puede ser un package por defecto, sin nombre
 - el archivo declara a qué package pertenece el código



visibilidad y acceso

- cuatro distinciones de visibilidad
`public`, `private`, `protected`, `package`
- un método se puede referir a:
 - los miembros privados de la clase a la que pertenece
 - miembros no privados de todas las clases del mismo `package`
 - miembros `protected` de superclases, en distintos `packages`
 - miembros `public` de clases en `packages` visibles, donde la visibilidad está determinada por el sistema de archivos
- nombres calificados (o usando *import*)
 - `java.lang.String.substring()`

package clase método

herencia

- semejante a smalltalk y C++
- las subclases heredan de las superclases
 - herencia simple únicamente – pero Java tiene interfaces
- algunas características adicionales
 - clases y métodos final (no se pueden heredar)

una subclase de ejemplo

```
class ColorPoint extends Point {  
    // métodos y campos adicionales  
    private Color c;  
    protected void setC (Color d)    {c = d;}  
    public Color  getC()              {return c;}  
    // se define el constructor  
    ColorPoint(int xval, Color cval) {  
        super(xval); // llama al constructor de Point  
        c = cval;    } // inicializa el campo ColorPoint  
};
```


clase *Object*

- todas las clases extienden otras clases
 - si no se explicita otra clase, la superclase es *Object*
- métodos de una clase *Object*
 - `getClass` – devuelve el objeto `Class` que representa la clase del objeto
 - `ToString` – devuelve la representación en string del objeto
 - `equals` – equivalencia de objetos por defecto (no de punteros)
 - `hashCode`
 - `Clone` – hace un duplicado de un objeto
 - `wait, notify, notifyAll` – para concurrencia
 - `finalize`

constructores y Super

- Java garantiza una llamada a constructor para cada objeto
- la herencia tiene que preservar esta propiedad
 - el constructor de subclase tiene que llamar al constructor de superclase
 - si el primer statement no es una llamada a super, el compilador inserta la llamada a super() automáticamente
 - si la superclase no tiene un constructor sin argumentos, causa un error de compilación
 - excepción: si un constructor llama a otro, entonces el segundo constructor es el responsable de llamar a super


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

se compila sin insertar la llamada a super

clases y métodos finales

- restringen herencia
 - las clases y métodos finales no se pueden redefinir, por ejemplo
`java.lang.String`
- para qué sirve
 - importante para seguridad
 - el programador controla el comportamiento de todas las subclases, crítico porque las subclases producen subtipos
 - si lo comparamos con virtual/non-virtual en C++, todo método es virtual hasta que se hace final

contenidos

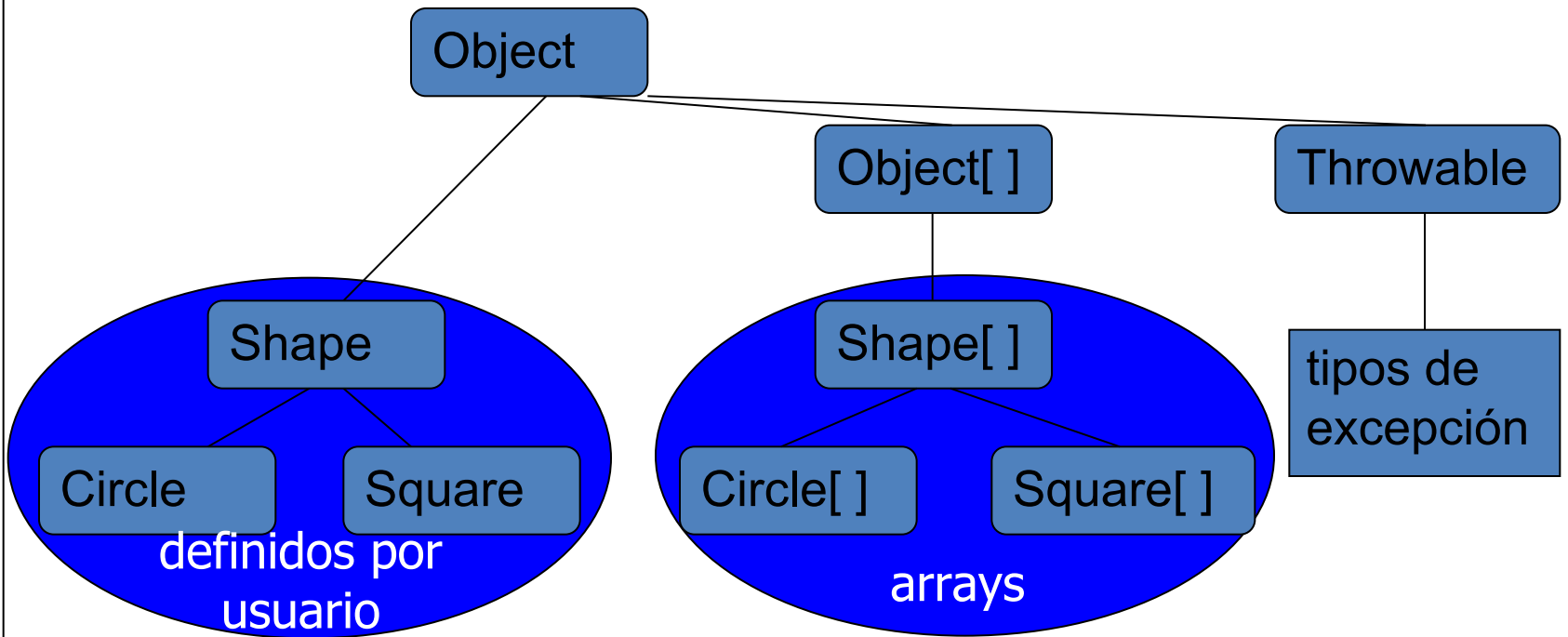
- objetos en Java
 - Clases, encapsulación, herencia
-  • sistema de tipos
 - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
 - básicos, wildcards, ...
- máquina virtual
 - Loader, verifier, linker, interpreter
 - Bytecodes para lookup de métodos
- temas de seguridad

tipos

- dos clases generales de tipos
 - tipos primitivos *que no son objetos*: enteros, booleanos
 - tipos de referencia: clases, interfaces, arrays
- chequeo estático de tipos
 - toda expresión tiene tipo, determinado por sus partes
 - algunas conversiones automáticas, muchos casteos se comprueban en tiempo de ejecución

clasificación de tipos de java

tipos de referencia



tipos primitivos



subtipado

- tipos primitivos
 - conversiones: int -> long, double -> long
- subtipado de clase semejante a C++
 - una subclase produce un subtipo
- Interfaces
 - clases completamente abstractas, sin implementación Clases con metodos pero sin el codigo de ellos
 - subtipado múltiple: una interfaz puede tener múltiples subtipos, que la implementan, la extienden

subtipado en interfaces: ejemplo

```
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 {  
    // no hereda ninguna implementación  
    // pero tiene que definir los métodos de Shape y Drawable
```

```
} Nos aseguramos que no hay nameClashes ya que solo se usa el nombre de un metodo y le damos nuestra  
implementacion.  
Esto sirve para asegurarme que todas las clases que implementa la interfaz, tiene esos metodos, es decir que me  
voy a poder comunicar con ellos. Basicamente sirve para organizacion
```

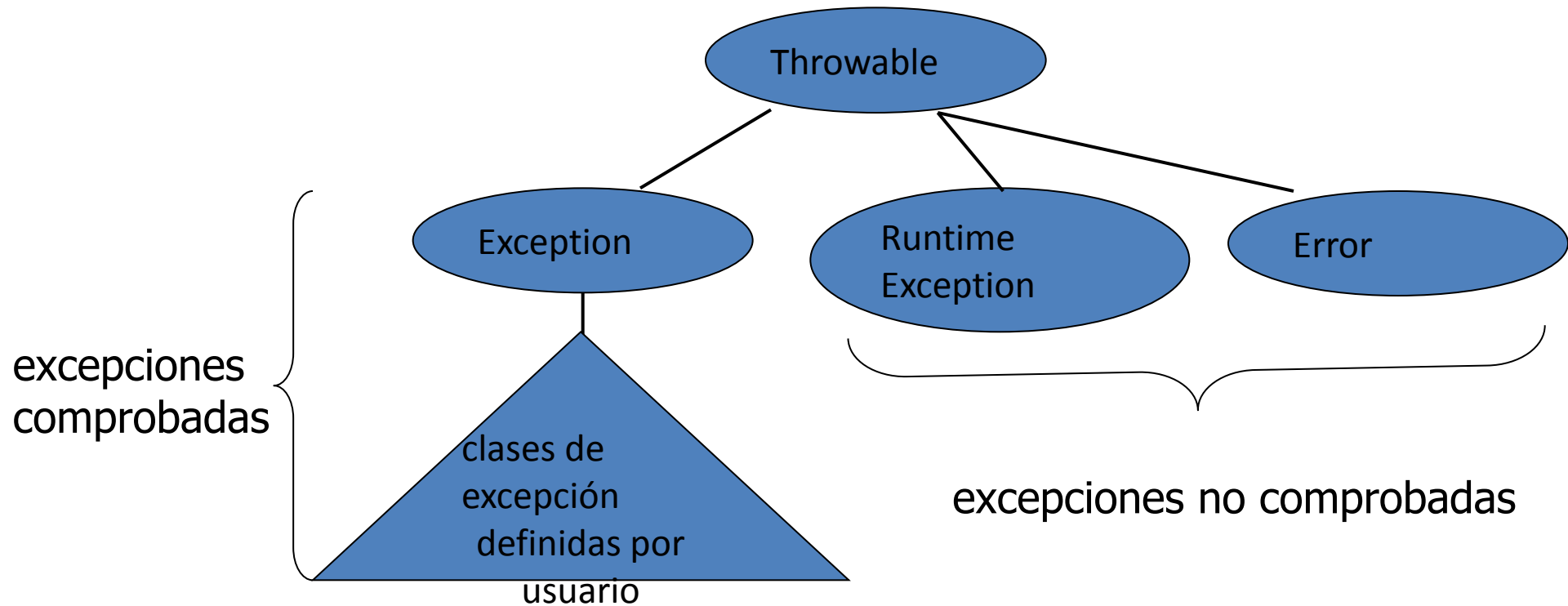

propiedades de las interfaces

- flexibilidad
 - permite un grafo de subtipado, en lugar de un árbol
 - evita problemas con herencia múltiple de implementaciones (como la herencia en diamante de C++)
- coste
 - no se conoce el offset en la tabla de consulta de métodos (*method lookup table*) en tiempo de compilación
 - hay diferentes bytecodes para consulta de métodos:
 - uno cuando se conoce la clase
 - otro cuando sólo se conoce la interfaz

excepciones

- funcionalidad semejante a otros lenguajes
 - construcciones para *throw* y *catch*
 - alcance dinámico
- algunas diferencias
 - una excepción es un objeto de una clase excepción
 - subtipado entre clases excepción
 - se usa subtipado para matchear el tipo de una excepción o pasarlo (semejante a ML)
 - el tipo de cada método incluye las excepciones que puede lanzar, todas subclases de *Exception*

clases **Exception**



si un método lanza una excepción comprobada, la excepción debe estar en el tipo del método

```
class WrongInputException extends Exception {
    WrongInputException(String s) {
        super(s);
    }
}
class Input {
    void method() throws WrongInputException {
        throw new WrongInputException("Wrong input");
    }
}
class TestInput {
    public static void main(String[] args){
        try {
            new Input().method();
        }
        catch(WrongInputException wie) {
            System.out.println(wie.getMessage());
        }
    }
}
```

bloques try / finally

- las excepciones se capturan en bloques `try`

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

por qué nuevos tipos de excepción?

- las excepciones pueden contener datos
 - la clase Throwable incluye un campo string para describir la causa de la excepción
 - se pasan otros datos declarando campos o métodos adicionales
- la jerarquía de subtipos se usa para capturar excepciones

`catch <exception-type> <identifier> { ... }`

captura cualquier excepción de cualquier subtipo y la liga al identificador

contenidos

- objetos en Java
 - Clases, encapsulación, herencia
- sistema de tipos
 - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
 - básicos, wildcards, ...
- máquina virtual
 - Loader, verifier, linker, interpreter
 - Bytecodes para lookup de métodos
- temas de seguridad



programación genérica

- la clase Object es supertipo de todos los tipos objeto
 - esto permite polimorfismo en objetos, porque se pueden aplicar las operaciones de la clase T a toda subclase $S <: T$
- Java 1.0 – 1.4 no tenían genéricos, y se consideró una gran limitación

ejemplo de construcción genérica: pila

- se pueden hacer pilas para cualquier tipo de objeto, y las operaciones asociadas a pila pila funcionan para cualquier tipo
- en C++ tendríamos la clase genérica `stack`

```
template <type t> class Stack {  
    private: t data;  Stack<t> * next;  
    public: void      push (t* x) { ... }  
              t*      pop   (      ) { ... }  
};
```

- qué se puede hacer en Java 1.0?

Java 1.0

vs genéricos

```
class Stack {  
    void push(Object o) {  
        ... }  
    Object pop() { ... }  
    ...}
```

```
String s = "Hello";  
Stack st = new Stack();  
...  
st.push(s);  
...  
s = (String) st.pop();
```

```
class Stack<A> {  
    void push(A a) { ... }  
    A pop() { ... }  
    ...}
```

```
String s = "Hello";  
Stack<String> st =  
    new Stack<String>();  
st.push(s);  
...  
s = st.pop();
```

por qué no se incorporan al principio?

- muchas distintas propuestas
- los objetivos básicos del lenguaje parecían cubiertos
- varios detalles que requieren esfuerzo
 - precisar exactamente las restricciones de tipado
 - implementación
 - en la virtual machine que ya existe?
 - bytecodes adicionales?
 - duplicar el código para cada instancia?
 - usar el mismo código, con casteos, para todas las instancias

la propuesta de la comunidad de Java (JSR 14)
se incorpora a Java 1.5

los genéricos de Java tienen comprobación de tipos

- una clase genérica usa operaciones en un tipo de parámetros
 - `PriorityQueue<T> ... if x.less(y) then ...`
- Dos posibles soluciones
 - C++: Linkear y fijarse si todas las operaciones se pueden resolver
 - Java: chequea tipos y compila los genéricos sin linkear

contenidos

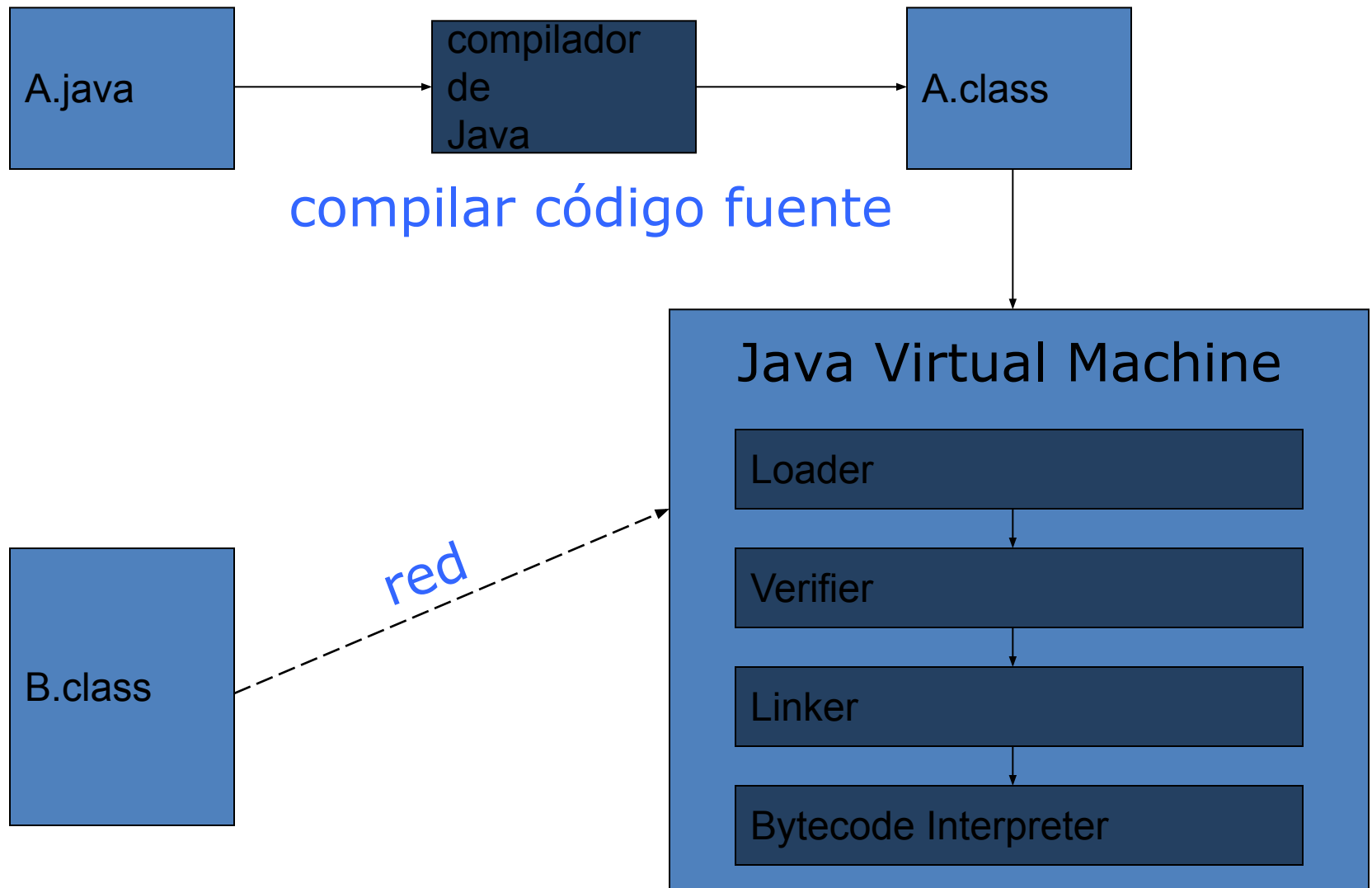
- objetos en Java
 - Clases, encapsulación, herencia
- sistema de tipos
 - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
 - básicos, wildcards, ...
- máquina virtual
 - Loader, verifier, linker, interpreter
 - Bytecodes para lookup de métodos
- temas de seguridad



implementación

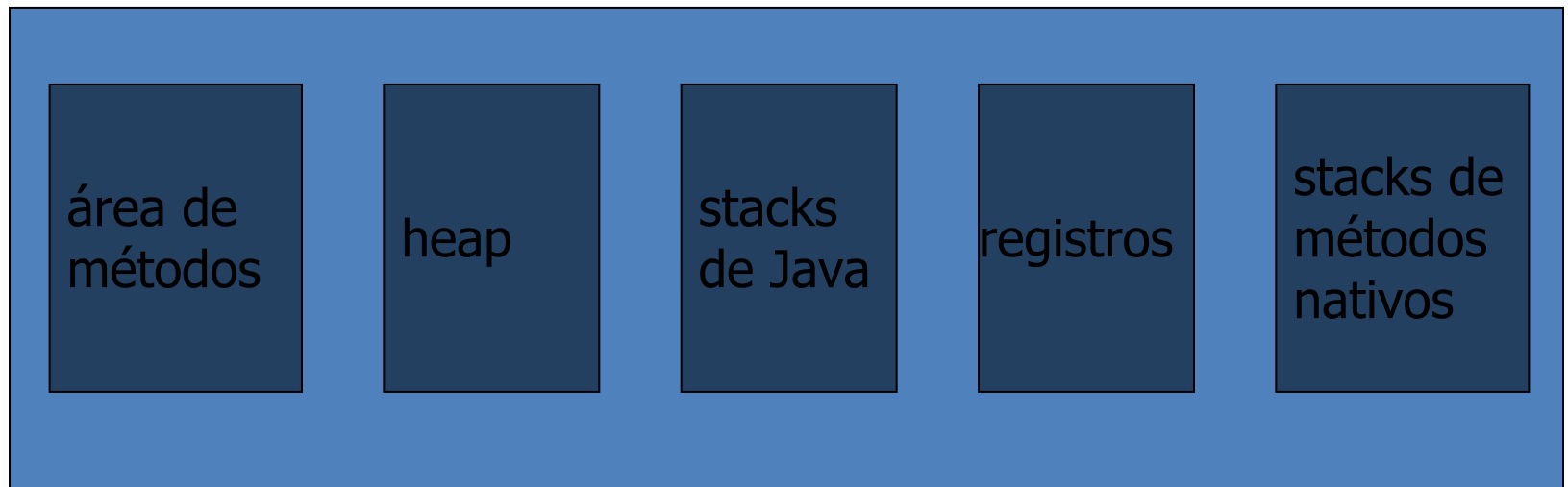
- compilador y máquina virtual
 - el compilador produce bytecode
 - la máquina virtual carga clases a demanda, verifica propiedades del bytecode e interpreta el bytecode
- por qué este diseño?
 - ya se habían usado intérpretes / compiladores de bytecode antes: Pascal, Smalltalk
 - minimizan la parte de la implementación dependiente de máquina
 - la optimización se hace en el bytecode
 - se mantiene muy simple el intérprete de bytecode
 - para Java, también aporta portabilidad
 - se puede transmitir el bytecode por la red

Arquitectura de la JVM



áreas de memoria de la JVM

- el programa en Java tiene uno o más threads
- cada thread tiene su propio stack
- todos los threads comparten el heap



carga de clases

- el sistema de ejecución carga las clases a medida que se necesitan
 - cuando se referencia una clase, el sistema de carga busca el archivo de instrucciones de bytecode compiladas
- el mecanismo de carga por defecto se puede sustituir definiendo otro objeto `ClassLoader`
 - se extiende la clase `ClassLoader`
 - `ClassLoader` no implementa el método abstracto `loadClass`, sino que tiene métodos que pueden usarse para implementar `loadClass`
 - se pueden obtener bytecodes de otra fuente

linker y verificador de la JVM

- Linker
 - añade la clase o interfaz compiladas al sistema de ejecución
 - crea los campos estáticos y los inicializa
 - resuelve nombres, reemplazándolos con referencias directas
- Verificador
 - comprueba el bytecode de una clase o interfaz antes de que se cargue
 - lanza la excepción `VerifyError`

Verifier

- Bytecode may not come from standard compiler
 - Evil hacker may write dangerous bytecode
- Verifier checks correctness of bytecode
 - Every instruction must have a valid operation code
 - Every branch instruction must branch to the start of some other instruction, not middle of instruction
 - Every method must have a structurally correct signature
 - Every instruction obeys the Java type discipline

Last condition is fairly complicated

Bytecode interpreter

- Standard virtual machine interprets instructions
 - Perform run-time checks such as array bounds
 - Possible to compile bytecode class file to native code
- Java programs can call native methods
 - Typically functions written in C
- Multiple bytecodes for method lookup
 - invokevirtual - when class of object known
 - invokeinterface - when interface of object known
 - invokestatic - static methods
 - invokespecial - some special cases

Type Safety of JVM

- Run-time type checking
 - All casts are checked to make sure type safe
 - All array references are checked to make sure the array index is within the array bounds
 - References are tested to make sure they are not null before they are dereferenced.
- Additional features
 - Automatic garbage collection
 - No pointer arithmetic

If program accesses memory, that memory is allocated to the program and declared with correct type

JVM uses stack machine

- **Java**

```
Class A extends Object {  
    int i  
    void f(int val) { i = val + 1;}  
}
```

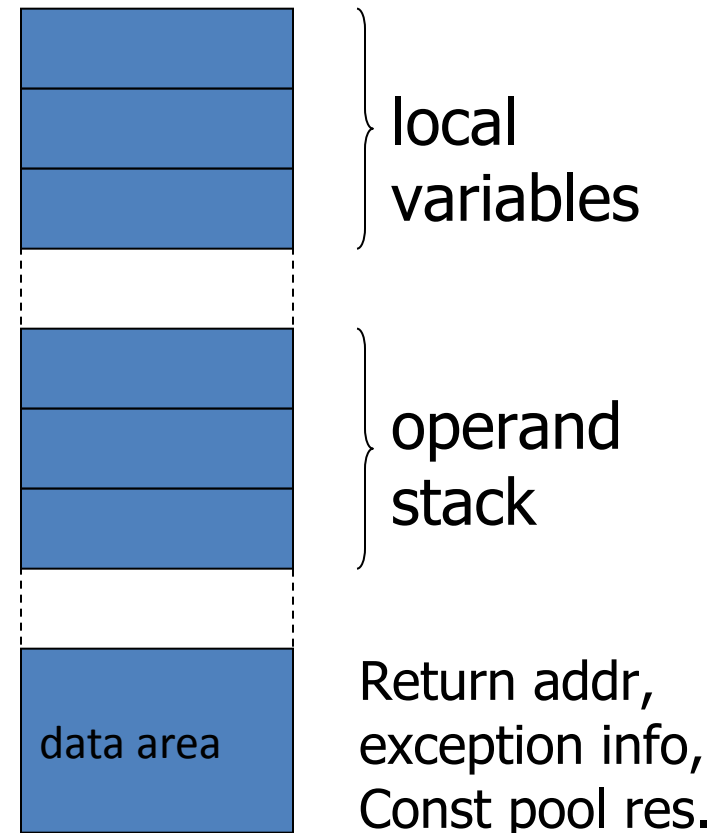
- **Bytecode**

```
Method void f(int)  
    aload 0    ; object ref this  
    iload 1    ; int val  
    iconst 1  
    iadd       ; add val +1  
    putfield #4 <Field int i>  
    return
```



refers to const pool

JVM Activation Record



Field and method access

- Instruction includes index into constant pool
 - Constant pool stores symbolic names
 - Store once, instead of each instruction, to save space
- First execution
 - Use symbolic name to find field or method
- Second execution
 - Use modified “quick” instruction to simplify search

invokeinterface <method-spec>

- Sample code

```
void add2(Incrementable x) { x.inc(); x.inc(); }
```

- Search for method

- find class of the object operand (operand on stack)
 - must implement the interface named in <method-spec>
- search the method table for this class
- find method with the given name and signature

- Call the method

- Usual function call with new activation record, etc.

Why is search necessary?

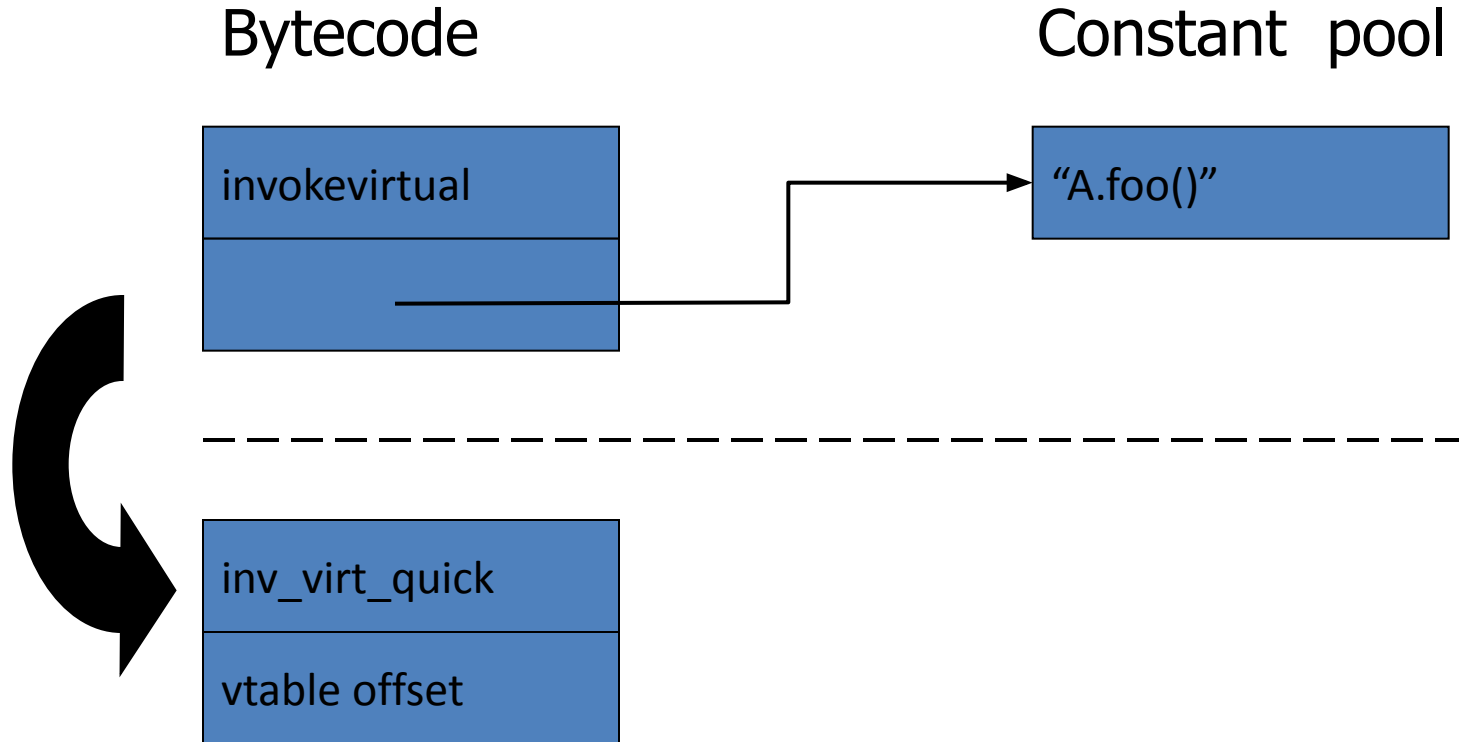
```
interface A {  
    public void f();  
}  
interface B {  
    public void g();  
}  
class C implements A, B {  
    ...;  
}
```

Class C cannot have method f first *and* method g first

invokevirtual <method-spec>

- Similar to invokeinterface, but class is known
- Search for method
 - search the method table of this class
 - find method with the given name and signature
- Can we use static type for efficiency?
 - Each execution of an instruction will be to object from subclass of statically-known class
 - Constant offset into vtable
 - like C++, but dynamic linking makes search useful first time
 - See next slide

Bytecode rewriting: invokevirtual

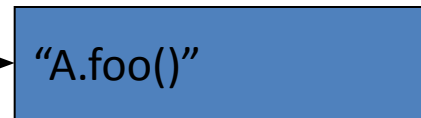
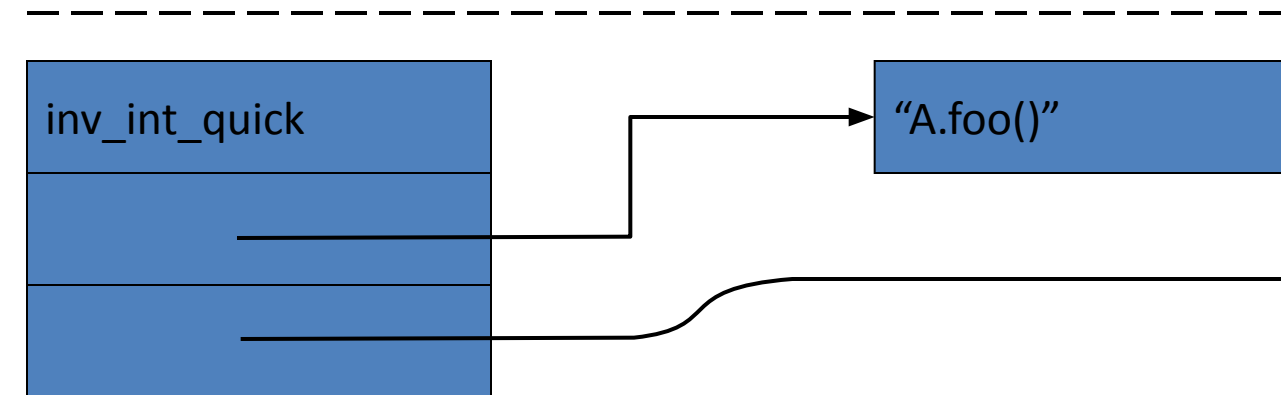
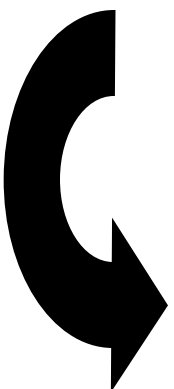
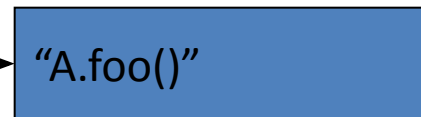
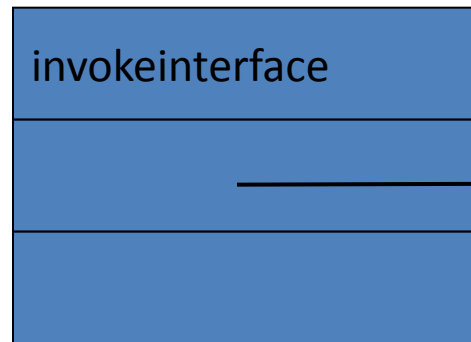


- After search, rewrite bytecode to use fixed offset into the vtable. No search on second execution.

Bytecode rewriting: invokeinterface

Bytecode

Constant pool



Cache address of method; check class on second use

Bytecode Verifier

- Let's look at one example to see how this works
- Correctness condition
 - No operations should be invoked on an object until it has been initialized
- Bytecode instructions
 - new <class> allocate memory for object
 - init <class> initialize object on top of stack
 - use <class> use object on top of stack
(idealization for purpose of presentation)

Object creation

- Example:

Point p = new Point(3) Java source

1: new Point

2: dup

3: iconst 3

4: init Point

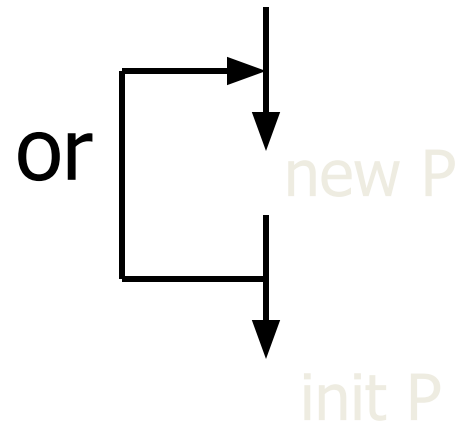
} bytecode

- No easy pattern to match
- Multiple refs to same uninitialized object
 - Need some form of alias analysis

Alias Analysis

- Other situations:

1: new P
2: new P
3: init P



- Equivalence classes based on line where object was created.

Tracking initialize-before-use

- Alias analysis uses line numbers
 - Two pointers to “unitialized object created at line 47” are assumed to point to same object
 - All accessible objects must be initialized before jump backwards (possible loop)
- Oversight in early treatment of local subroutines
 - Used in implementation of **try-finally**
 - Object created in **finally** not necessarily initialized
- No clear security consequence
 - Bug fixed

Have proved correctness of modified verifier for init

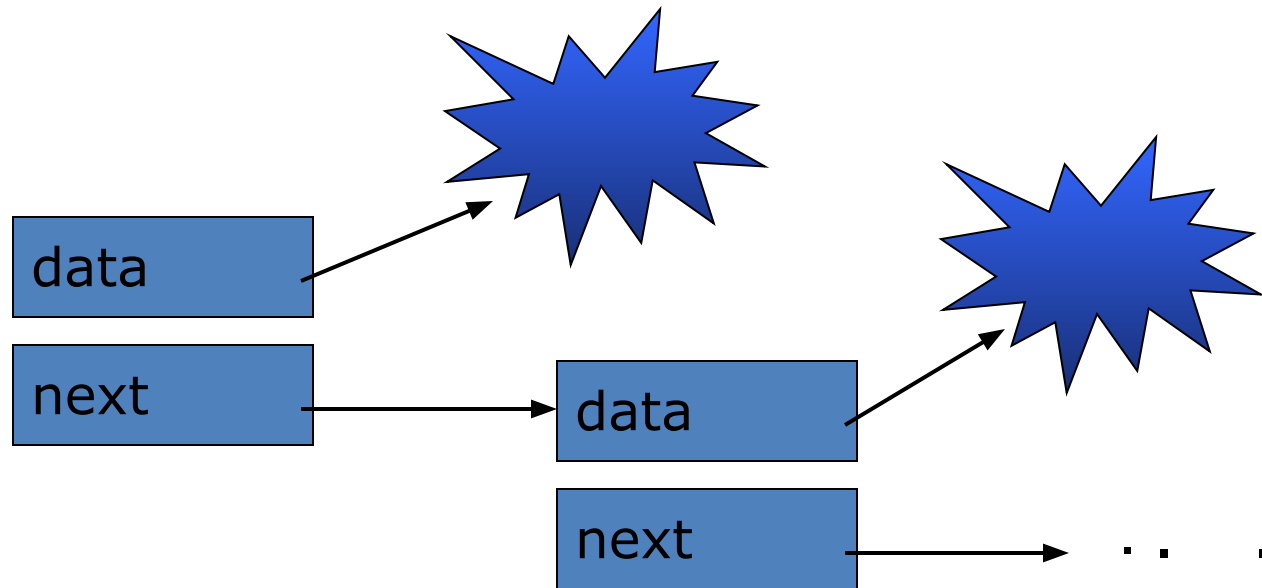
Implementing Generics

- Two possible implementations
 - Heterogeneous: instantiate generics
 - Homogeneous: translate generic class to standard class

- Example for next few slides: generic list class

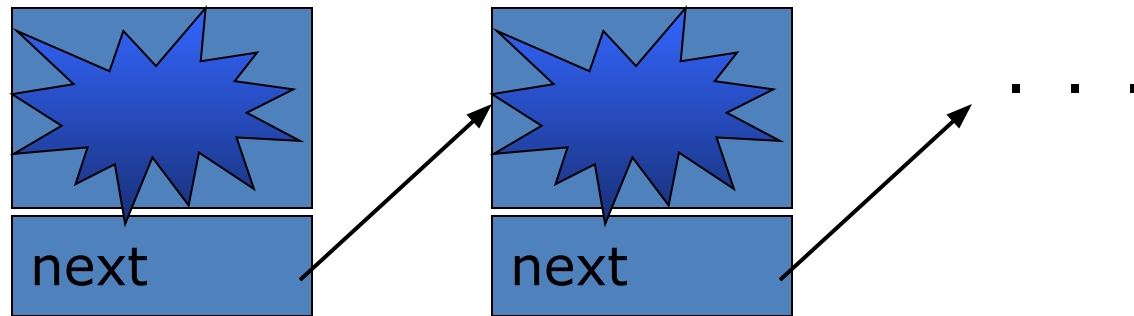
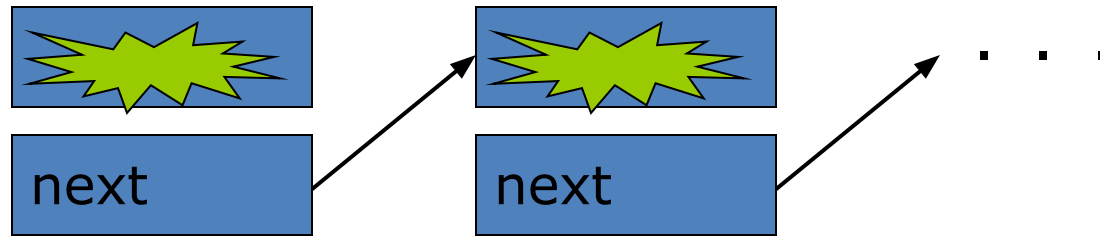
```
template <type t> class List {  
    private: t* data; List<t> * next;  
    public: void    Cons (t* x) { ... }  
             t*     Head (    ) { ... }  
             List<t> Tail (    ) { ... }  
};
```

“Homogeneous Implementation”



Same representation and code for all types of data

“Heterogeneous Implementation”



Specialize representation, code according to type

Issues

- Data on heap, manipulated by pointer (Java)
 - Every list cell has two pointers, data and next
 - All pointers are same size
 - Can use same representation, code for all types
- Data stored in local variables (C++)
 - List cell must have space for data
 - Different representation for different types
 - Different code if offset of fields built into code
- When is template instantiated?
 - Compile- or link-time (C++)
 - Java alternative: class load time – next few slides
 - Java Generics: no “instantiation”, but erasure at compile time
 - C# : just-in-time instantiation, with some code-sharing tricks ...

Heterogeneous Implementation for Java

- Compile generic class `C<param>`
 - Check use of parameter type according to constraints
 - Produce extended form of bytecode class file
 - Store constraints, type parameter names in bytecode file
- Expand when class `C<actual>` is loaded
 - Replace parameter type by actual class
 - Result is ordinary class file
 - This is a preprocessor to the class loader:
 - No change to the virtual machine
 - No need for additional bytecodes

A heterogeneous implementation is possible, but was not adopted for standard

Example: Hash Table

```
interface Hashable {  
    int hashCode ();  
};
```

```
class HashTable < Key implements Hashable, Value> {  
    void    Insert (Key k, Value v) {  
        int bucket = k.hashCode();  
        InsertAt (bucket, k, v);  
    }  
    ...  
};
```

Generic bytecode with placeholders

```
void Insert (Key k, Value v) {  
    int bucket = k.HashCode();  
    InsertAt (bucket, k, v);  
}
```

```
Method void Insert($1, $2)  
    aload_1  
    invokevirtual #6 <Method $1.HashCode()I>  
    istore_3    aload_0    iload_3    aload_1    aload_2  
    invokevirtual #7 <Method HashTable<$1,$2>.  
                        InsertAt(IL$1;L$2;)V>  
    return
```

Instantiation of generic bytecode

```
void Insert (Key k, Value v) {  
    int bucket = k.GetHashCode();  
    InsertAt (bucket, k, v);  
}
```

```
Method void Insert(Name, Integer)  
    aload_1  
    invokevirtual #6 <Method Name.GetHashCode()I>  
    istore_3    aload_0    iload_3    aload_1    aload_2  
    invokevirtual #7 <Method  
    HashTable<Name,Integer>  
                    InsertAt(ILName;LInteger;)V>  
    return
```

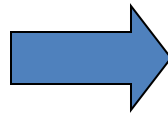

Loading parameterized class file

- Use of `HashTable <Name, Integer>` invokes loader
- Several preprocess steps
 - Locate bytecode for parameterized class, actual types
 - Check the parameter constraints against actual class
 - Substitute actual type name for parameter type
 - Proceed with verifier, linker as usual
- Can be implemented with ~500 lines Java code
 - Portable, efficient, no need to change virtual machine

Java 1.5 Implementation

- Homogeneous implementation

```
class Stack<A> {  
    void push(A a) { ... }  
    A pop() { ... }  
    ...}
```



```
class Stack {  
    void push(Object o) { ... }  
    Object pop() { ... }  
    ...}
```

- Algorithm
 - replace class parameter <A> by Object, insert casts
 - if <A extends B>, replace A by B
- Why choose this implementation?
 - Backward compatibility of distributed bytecode
 - Surprise: sometimes faster because class loading slow

Some details that matter

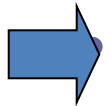
- Allocation of static variables
 - Heterogeneous: separate copy for each instance
 - Homogenous: one copy shared by all instances
- Constructor of actual class parameter
 - Heterogeneous: `class G<T> ... T x = new T;`
 - Homogenous: `new T` may just be `Object` !
 - Creation of new object is not allowed in Java
- Resolve overloading
 - Heterogeneous: resolve at instantiation time (C++)
 - Homogenous: no information about type parameter

Example

- This Code is not legal java
 - class C<A> { A id (A x) {...} }
 - class D extends C<String> {
 Object id(Object x) {...}
}
- Why?
 - Subclass method looks like a different method, but after erasure the signatures are the same

contenidos

- objetos en Java
 - Clases, encapsulación, herencia
- sistema de tipos
 - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
 - básicos, wildcards, ...
- máquina virtual
 - Loader, verifier, linker, interpreter
 - Bytecodes para lookup de métodos



temas de seguridad

seguridad en Java

- seguridad
 - evitar uso no autorizado de recursos computacionales
 - seguridad en Java
 - el código Java puede leer input de usuarios despistados o atacantes maliciosos
 - el código Java se puede transmitir por la red
- Java está diseñado para reducir riesgos de seguridad

mecanismos de seguridad

- **Sandboxing** (jugar en el arenero)
 - el programa se ejecuta en un entorno restringido
 - se aplica a:
 - características del loader, verificador, e intérprete que restringen al programa
 - Java Security Manager, un objeto especial que ejerce control de acceso
- **firma de código**
 - se usan principios criptográficos para establecer el origen de un archivo de clase
 - la usa el security manager

ataque de Buffer Overflow

- es el problema de seguridad más frecuente
- en general, basado en red:
 - el atacante envía mensajes de red diseñados especialmente
 - el input hace que un programa con privilegios (por ej., Sendmail) haga algo que no tenía que hacer
- no funciona en Java!

ejemplo de código en C para ataque de buffer overflow

```
void f (char *str) {  
    char buffer[16];  
    ...  
    strcpy(buffer, str);  
}  
void main() {  
    char large_string[256];  
    int i;  
    for( i = 0; i < 255;  
        i++)  
        large_string[i] = 'A';  
    f(large_string);  
}
```

- la función
 - copia str a un buffer hasta que se encuentra el caracter nulo
 - podría escribir hasta pasado el final del buffer, por encima de la dirección de retorno de la función!!
- la llamada
 - escribe 'A' sobre el activation record de f
 - la función “retorna” a la ubicación 0x4141414141
 - esto causa un segmentation fault
- variaciones
 - poner una dirección con significado en el string
 - poner código en el string y saltar ahí!

para saber más: *Smashing the stack for fun and profit*

Java Sandbox

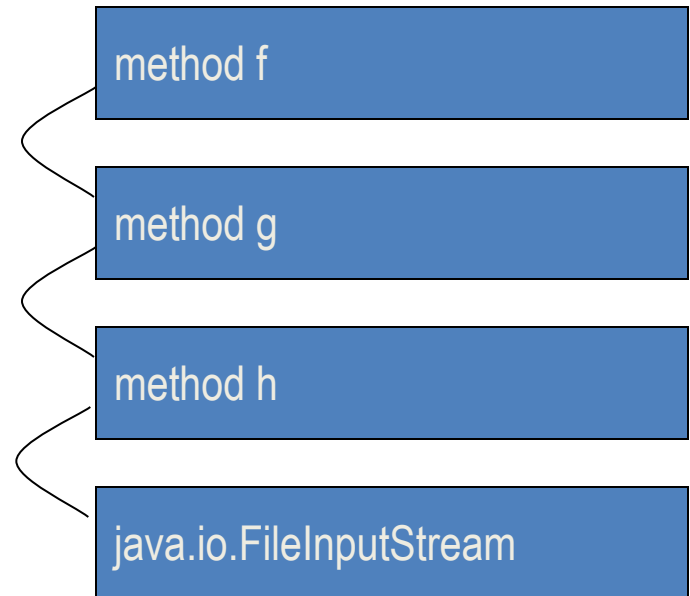
- Class loader
 - namespaces distintos para distintos class loaders
 - asocia un *protection domain* con cada clase
- tests en tiempo de ejecución del Verifier y JVM
 - no se permiten casteos sin comprobación de tipos ni otros errores de tipo, no se permite array overflow
 - preserva los niveles de visibilidad private y protected
- Security Manager
 - lo llaman las funciones para decidir si deben hacer lugar a un pedido
 - usa el *protection domain* asociado al código y política de usuario

Security Manager

- las funciones de la biblioteca de Java llaman al security manager
- respuesta en tiempo de ejecución
 - decide si el código que llama tiene permiso para hacer la operación
 - examinar el dominio de protección de la clase que llama
 - Signer: organización que firmó el código antes de cargarlo
 - Ubicación: URL de donde vienen las clases
 - da permiso de acceso según la política del sistema

inspección del stack

- el permiso depende de:
 - permiso del método que llama
 - permiso de todos los métodos por encima de él en el stack, hasta llegar a un método confiable



Stories: Netscape font / passwd bug; Shockwave plug-in

ejemplos de métodos del Security Manager

checkExec	comprueba si los comandos de sistema se pueden ejecutar.
checkRead	comprueba si un archivo se puede leer.
checkWrite	comprueba si un archivo se puede escribir.
checkListen	comprueba si un puerto determinado se puede escuchar.
checkConnect	comprueba si se puede crear una conexión de red.
checkCreateClassLoader	comprueba para evitar que se instalen más ClassLoaders.

resumen

- objetos
 - tienen campos y métodos
 - alojados en el heap, se acceden con punteros, con recolección de basura
- clases
 - Public, Private, Protected, Package (no exactamente como en C++)
 - pueden tener miembros estáticos (propios de la clase)
 - Constructores y métodos finalize
- herencia
 - herencia simple
 - métodos y clases finales (no pueden tener hijas)

resumen

- subtipado
 - determinado por la jerarquía de herencia
 - una clase puede implementar muchas interfaces
- Virtual machine
 - carga bytecode para clases en tiempo de ejecución
 - el verificador comprueba el bytecode
 - el intérprete también hace comprobaciones en tiempo de ejecución
 - casteos
 - límites de arreglos
- portabilidad y seguridad

Some Highlights

- Dynamic lookup
 - Different bytecodes for by-class, by-interface
 - Search vtable + Bytecode rewriting or caching
- Subtyping
 - Interfaces instead of multiple inheritance
 - Awkward treatment of array subtyping (my opinion)
- Generics
 - Type checked, not instantiated, some limitations (`<T>...new T`)
- Bytecode-based JVM
 - Bytecode verifier
 - Security: security manager, stack inspection

Comparison with C++

- Almost everything is object + Simplicity - Efficiency
 - except for values from primitive types
- Type safe + Safety +/- Code complexity - Efficiency
 - Arrays are bounds checked
 - No pointer arithmetic, no unchecked type casts
 - Garbage collected
- Interpreted + Portability + Safety - Efficiency
 - Compiled to byte code: a generalized form of assembly language designed to interpret quickly.
 - Byte codes contain type information

Comparison

(cont'd)

- Objects accessed by ptr + Simplicity - Efficiency
 - No problems with direct manipulation of objects
- Garbage collection: + Safety + Simplicity - Efficiency
 - Needed to support type safety
- Built-in concurrency support + Portability
 - Used for concurrent garbage collection (avoid waiting?)
 - Concurrency control via synchronous methods
 - Part of network support: download data while executing
- Exceptions
 - As in C++, integral part of language design