

Programación orientada a objetos: Java

Paradigmas de la Programación
FaMAF-UNC 2015

capítulo 13

basado en filminas de John Mitchell

contenidos

- hisotria y objetivos
- clases y herencia
- tipos y subtipado
- genéricos
- máquina virtual
- lookup de métodos
- análisis de verificador
- implementación de genéricos
- seguridad

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

historia de releases

- 1995 (1.0) – primer release público
- 1997 (1.1) – clases internas
- 2001 (1.4) – aserciones
 - verificar la comprensión que tiene el programador del código
- 2004 (1.5) – Tiger
 - genéricos, foreach, y muchos otros

<http://java.sun.com/developer/technicalArticles/releases/j2se15/>
mejoras a través del Java Community Process

mejoras en el JDK 5 (= Java 1.5)

- genéricos
 - polimorfismo y seguridad en tiempo de ejecución (JSR 14)
- ciclo for aumentado
 - para iterar sobre colecciones y arreglos (JSR 201)
- Autoboxing/Unboxing
 - conversión automática de tipos wrapper a primitivos (JSR 201)
- enums seguras en tipos
 - tipos enumerados Enumerated types with arbitrary methods and fields (JSR 201)
- Varargs
 - Puts argument lists into an array; variable-length argument lists
- Static Import
 - Avoid qualifying static members with class names (JSR 201)
- Annotations (Metadata)
 - Enables tools to generate code from annotations (JSR 175)
- Concurrency utility library, led by Doug Lea (JSR-166)

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;}    //  
    constructor  
};
```

inicialización de objetos

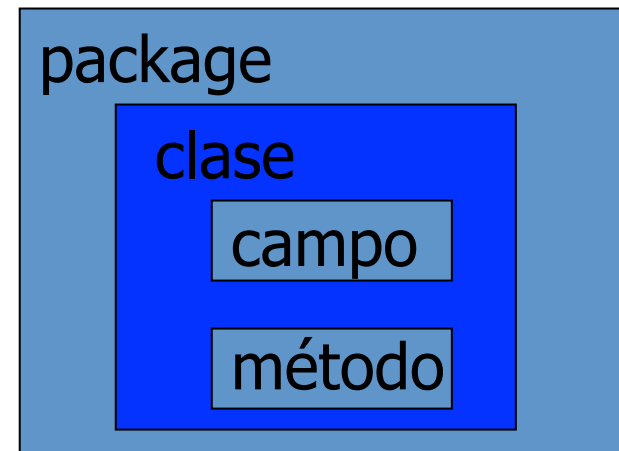
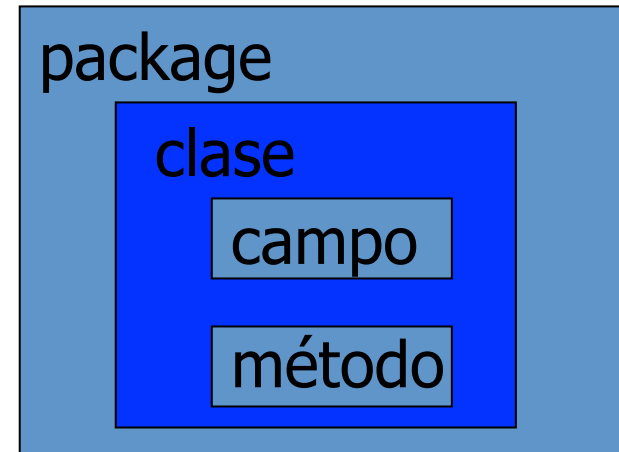
- Java garantiza la llamada al constructor para cada objeto
 - se aloja memoria
 - se llama al constructor para inicializar memoria
- los campos estáticos de la clase se inicializan en tiempo de carga

Garbage Collection y *Finalize*

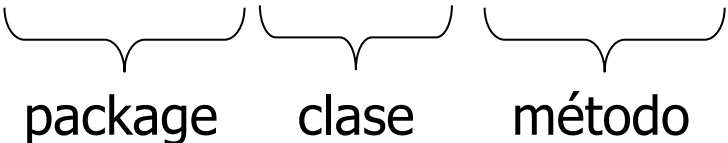
- los objetos pasan por la recolección de basura
 - no hay una instrucción *free* explícita
 - evita punteros colgantes
- problema
 - qué pasa si un objeto ha abierto un archivo o tiene un lock?
- solución
 - método *finalize*, llamado por el garbage collector

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

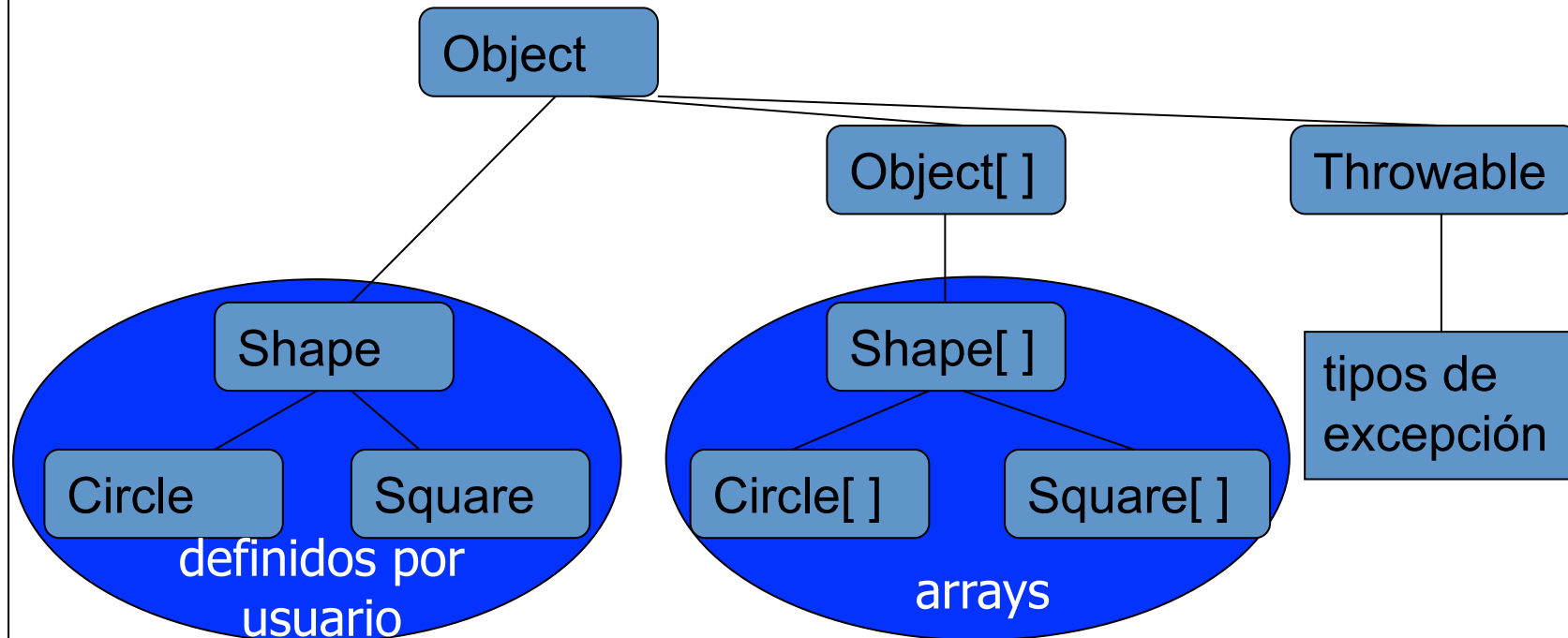
- objetos en Java
  - Clases, encapsulación, herencia
- ➡ sistema de tipos
  - tipos primitivos, interfaces, arreglos, excepciones
- genéricos (añadidos en Java 1.5)
  - básicos, wildcards, ...
- máquina virtual
  - Loader, verifier, linker, interpreter
  - Bytecodes para lookup de métodos
- temas de 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
  - ejemplo, si tenemos  $A <: B$ 
    - si  $A \ x$ , se puede usar  $x$  como argumento para un método que requiere  $B$
    - si  $B \ x$ , entonces podemos intentar castear a  $A$
    - el casteo se comprueba en tiempo de ejecución, y puede levantar una excepción

# clasificación de tipos de java

tipos de referencia



tipos primitivos

boolean

int

byte

...

float

long



# subtipado

- tipos primitivos
  - conversiones: int -> long, double -> long, ...
- subtipado de clase semejante a C++
  - una subclase produce un subtipo
  - herencia simple => las subclases forman un árbol
- Interfaces
  - clases completamente abstractas, sin implementación
  - subtipado múltiple: una interfaz puede tener múltiples subtipos, que la implementan, la extienden
- Arrays
  - subtipado covariante

# subtipado en clases

- conformidad de la signatura
  - las signaturas de las subclases tienen que conformar a la de la superclase
- formas conformantes en que puede variar la signatura:
  - tipos de los argumentos
  - tipo del resultado
  - excepciones
  - en Java 1.1, los argumentos y resultados debían tener tipos idénticos
  - en Java 1.5 se permite especialización de tipo covariante

# 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
}
```

# 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

# tipos arreglo

- Automatically defined
  - Array type `T[ ]` exists for each class, interface type `T`
  - Cannot extended array types (array types are final)
  - Multi-dimensional arrays are 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
  - if  $S <: T$  then  $S[ ] <: T[ ]$
- 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
```

# Covariance problem again ...

- Remember Simula problem
  - If  $A \leq B$ , then  $A \text{ ref} \leq B \text{ ref}$
  - Needed run-time test to prevent bad assignment
  - Covariance for assignable cells is not right in principle
- Explanation
  - interface of “T reference cell” is
    - put :  $T \rightarrow T \text{ ref}$
    - get :  $T \text{ ref} \rightarrow T$
  - Remember covariance/contravariance of functions

# Afterthought on Java arrays

Date: Fri, 09 Oct 1998 09:41:05 -0600

From: bill joy

Subject: ...[discussion about java genericity]

actually, java array covariance was done for less noble reasons ...: it made some generic "bcopy" (memory copy) and like operations much easier to write...

I proposed to take this out in 95, but it was too late (...).

i think it is unfortunate that it wasn't taken out...

it would have made adding genericity later much cleaner, and [array covariance] doesn't pay for its complexity today.

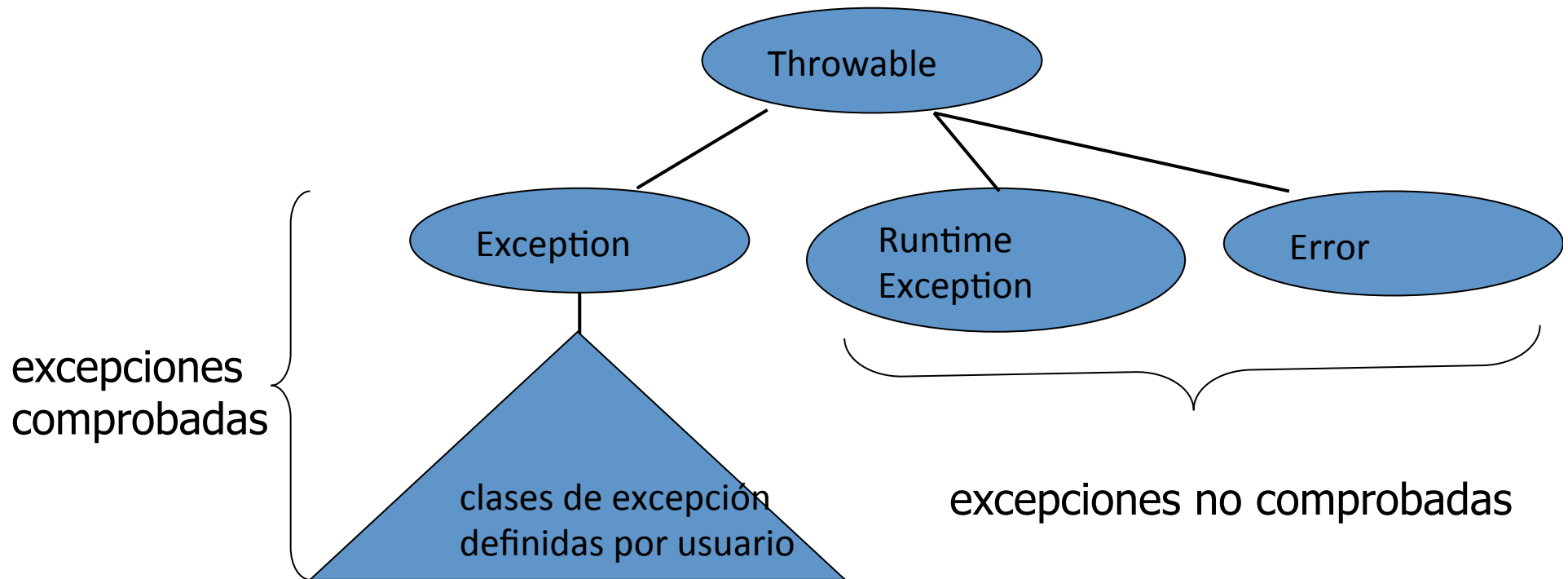
wnj



# 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());
 }
 }
}
```

# Try/finally blocks

- 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  $S \leq T$
- 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

# JSR 14 Java Generics (Java 1.5, “Tiger”)

- Adopts syntax on previous slide
- Adds auto boxing/unboxing

User conversion

```
Stack<Integer> st =
 new Stack<Integer>();
st.push(new Integer(12));
...
int i = (st.pop()).intValue();
```

conversion

Automatic

```
Stack<Integer> st =
 new Stack<Integer>();
st.push(12);
...
int i = st.pop();
```

# los genéricos de Java tienen comprobación de tipos

- una clase genérica usa operaciones en un tipo de parámetros
  - Example: `PriorityQueue<T> ...` if `x.less(y)` then ...
- Two possible solutions
  - C++: Link and see if all operations can be resolved
  - Java: Type check and compile generics w/o linking
    - May need additional information about type parameter
      - What methods are defined on parameter type?
      - Example: `PriorityQueue<T extends ...>`

# Example

- Generic interface

```
interface Collection<A> {
 public void add (A x);
 public Iterator<A> iterator ();
}
```

```
interface Iterator<E> {
 E next();
 boolean hasNext();
}
```

- Generic class implementing Collection interface

```
class LinkedList<A> implements Collection<A> {
 protected class Node {
 A elt;
 Node next = null;
 Node (A elt) { this.elt = elt; }
 }
 ...
}
```

# Wildcards

- Example

```
void printElements(Collection<?> c) {
 for (Object e : c)
 System.out.println(e);
}
```

- Meaning: Any representative from a family of types

- unbounded wildcard    ?
  - all types
- lower-bound wildcard    ? extends Supertype
  - all types that are subtypes of Supertype
- upper-bound wildcard    ? super Subtype
  - all types that are supertypes of Subtype

# Type concepts for understanding Generics

- Parametric polymorphism

–  $\text{max} : \forall t \underbrace{((t \times t) \rightarrow \text{bool})}_{\text{given lessThan function}} \rightarrow \underbrace{((t \times t) \rightarrow t)}_{\text{return max of two arguments}}$

- Bounded polymorphism

–  $\text{printString} : \forall t \underbrace{<: \text{Printable}}_{\text{for every subtype } t \text{ of Printable}} . \underbrace{t \rightarrow \text{String}}_{\text{function from } t \text{ to String}}$

- F-Bounded polymorphism

–  $\text{max} : \forall t \underbrace{<: \text{Comparable}}_{\text{for every subtype } t \text{ of Comparable}} (t) \rightarrow \underbrace{t \times t \rightarrow t}_{\text{return max of two arguments}}$

# F-bounded subtyping

- Generic interface

```
interface Comparable<T> { public int compareTo(T arg); }
```

- $x.compareTo(y)$  = negative, 0, positive if  $y$  is  $< = >$   $x$

- Subtyping

```
interface A { public int compareTo(A arg);
 int anotherMethod (A arg); ... }
```

<:

```
interface Comparable<A> { public int compareTo(A arg); }
```



# Example static max method

- Generic interface

```
interface Comparable<T> { public int compareTo(T arg); ... }
```

- Example

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {
 T candidate = coll.iterator().next();
 for (T elt : coll) {
 if (candidate.compareTo(elt) < 0) candidate = elt;
 }
 return candidate;
}
```

candidate.compareTo :  $T \rightarrow \text{int}$

# This would typecheck without F-bound ...

- Generic interface

```
interface Comparable<T> { public int compareTo(T arg); ... }
```

Object

- Example

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {
 T candidate = coll.iterator().next();
 for (T elt : coll) {
 if (candidate.compareTo(elt) < 0) candidate = elt;
 }
 return candidate;
}
```

candidate.compareTo : T -> int

Object

How could you write an implementation of this interface?

# Generics are *not* co/contra-variant

- Array example (review)

```
Integer[] ints = new Integer[] {1,2,3};
```

```
Number[] nums = ints;
```

```
nums[2] = 3.14; // array store -> exception at run
time
```

- List example

```
List<Integer> ints = Arrays.asList(1,2,3);
```

```
List<Number> nums = ints; // compile-time error
```

– Second does not compile because

```
List<Integer> <: List<Number>
```

# Return to wildcards

- Recall example

```
void printElements(Collection<?> c) {
 for (Object e : c)
 System.out.println(e);
}
```

- Compare to

```
void printElements(Collection<Object> c) {
 for (Object e : c)
 System.out.println(e);
}
```

- This version is *much* less useful than the old one
  - Wildcard allows call with kind of collection as a parameter,
  - Alternative only applies to Collection<Object>, not a supertype of other kinds of collections!

# Implementing Generics

- Type erasure
  - Compile-time type checking uses generics
  - Compiler eliminates generics by erasing them
    - Compile `List<T>` to `List`, `T` to `Object`, insert casts
- “Generics are not templates”
  - Generic declarations are typechecked
  - Generics are compiled once and for all
    - No instantiation
    - No “code bloat”

More later when we talk about virtual machine ...

## Additional links for material not in book

- Enhancements in JDK 5
  - <http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html>
- J2SE 5.0 in a Nutshell
  - <http://java.sun.com/developer/technicalArticles/releases/j2se15/>
- Generics
  - <http://www.langer.camelot.de/Resources/Links/JavaGenerics.htm>