Clases de Utilidad y Excepciones

Manuel Freire Morán
Iván Martínez Ortiz

Facultad de Informática
Universidad Complutense

Idea general

- Este capítulo da una idea de la **librería estándar**
  - StringBuilder, StringTokenizer y expresiones regulares para manipulación de cadenas
  - Date y calendarios para manipular fechas (y SimpleDateFormat para mostrarlas y leerlas).
  - Para trabajar con números, NumberFormat permite hacer entrada/salida eficiente; y BigInteger/Decimal no tener nunca problemas de precisión.
  - Los contenedores genéricos de Collections son imprescindibles para estructurar datos
- La librería estándar usa **excepciones** para señalizar errores o situaciones excepcionales
- Hay **interfaces estándar** y métodos de Object que es importante conocer para hacer buen uso de contenedores.
- Las **anotaciones** permiten extender el lenguaje, automatizando tareas repetitivas.

Iván Martínez Ortiz / Manuel Freire Morán

Excepciones

```
public Circulo(Punto centro, int radio) {
    if (radio < 0) {
        throw new IllegalArgumentException(
            "radio negativo");
    }
    this.centro = centro;
    this.radio = radio;
} // lanzamiento
```

```
Circulo c = null;
try {
    c = new Circulo(new Punto(0,0), -1);
} catch (IllegalArgumentException iae) {
    System.err.println(iae.getMessage());
    iae.printStackTrace();
} // manejo
```

Permiten

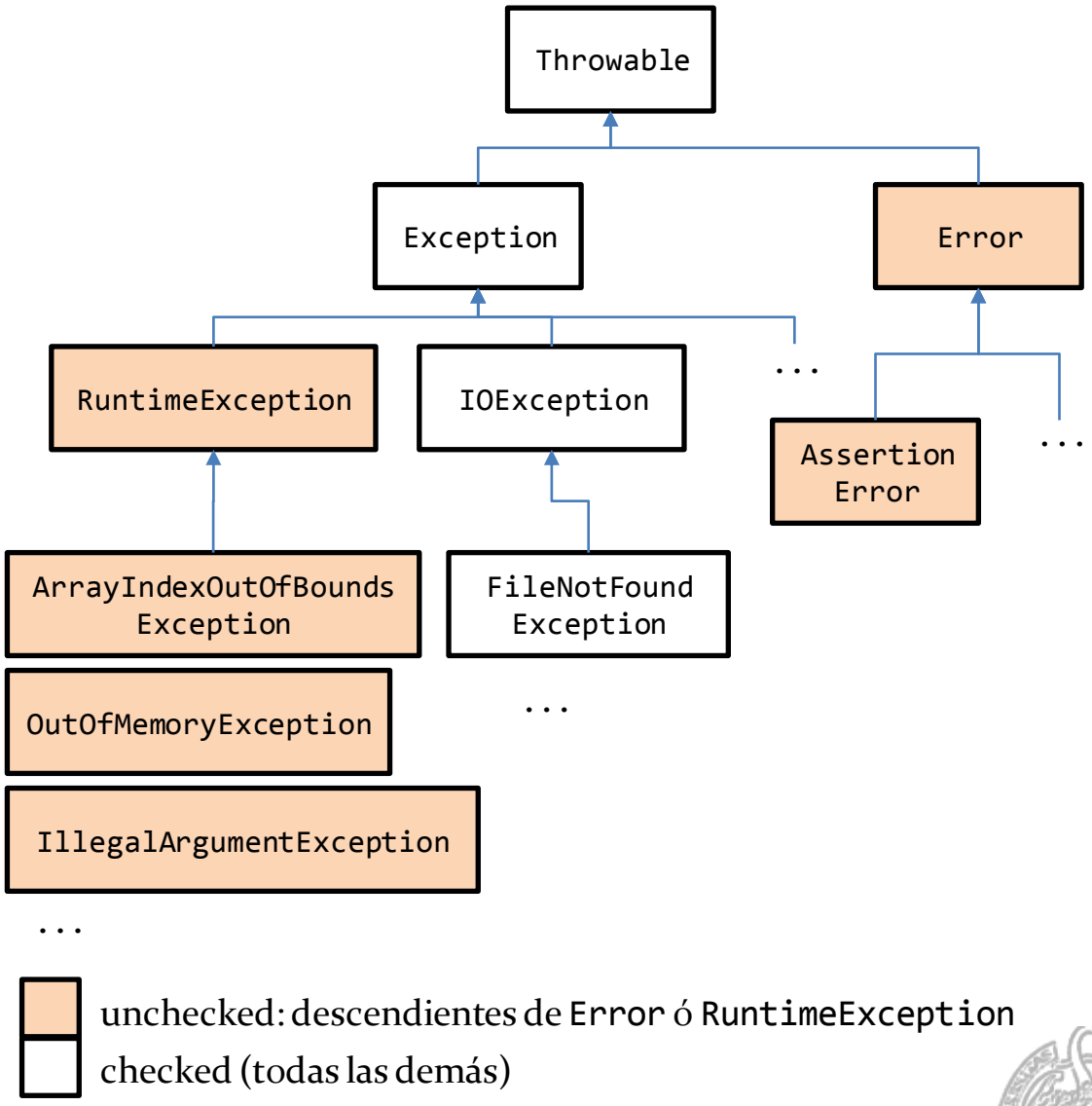
- Avisar de que se *puede* producir un error (ejemplo: java.io.IOException)
- Almacenar/consultar las causas de los errores
- Gestionar errores, sin llenar el código de comprobaciones de error

Iván Martínez Ortiz / Manuel Freire Morán

Exception vs Error/RuntimeException

Dos tipos

- declaradas: ("*checked*")
  - *tienes* que declararlas
  - *tienes* que manejarlas ó delegarlas explicitamente
- implícitas ("*unchecked*")
  - no declarables
  - no tienes porqué manejarlas



Iván Martínez Ortiz / Manuel Freire Morán

Algunas excepciones estándares

- Declaradas (imprescindible manejarlas ó propagarlas explícitamente)
  - Errores de entrada/salida: IOException
    - Fichero no existe: FileNotFoundException
    - Fin-de-fichero: EOFException

  …
- No-declaradas (opcional manejarlas, no puedes propagarlas)
  - Errores en argumentos: IllegalArgumentException
  - Intentando mirar dentro de null: NullPointerException
  - Error convirtiendo cadena a numero: NumberFormatException
  - Sin memoria: OutOfMemoryException
  - Índice de array no válido: ArrayIndexOutOfBounds

  …

Iván Martínez Ortiz / Manuel Freire Morán

Manejando una excepción

```
File f = null;
try {
    f = new File("hola.txt");
    f.delete();
} catch (FileNotFoundException fnfe) {
    // fnfe es una excepción de tipo FileNotFoundException
} catch (IOException ioe) {
    // ioe es una IOException, pero no FileNotFoundException
} catch (Exception e) {
    // e no es ni FileNotFoundException ni IOException
} finally {
    // codigo que se ejecuta SIEMPRE, haya o no excepción
}
```

- La estructura try { … } catch (*Tipo nombre*) { … } finally { …} permite capturar excepciones
- Es posible *omitir* los bloques catch y finally (pero sólo se considera que has tratado un tipo de excepción si tienes un catch para ella).
  - OJO: la falta de finally provoca errores (e.g. no se cierra el fichero al capturar la excepción)

Iván Martínez Ortiz / Manuel Freire Morán

Contenido de una excepción

```
// ejemplo de hacer un printStackTrace() sobre una NullPointerException
in thread "main" java.lang.NullPointerException
        at ucm.gea.demo.Libro.getTitulo(Libro.java:16)
        at ucm.gea.demo.Autor.getLibros(Autor.java:25)
        at ucm.gea.demo.Main.main(Main.java:14)
```

- Toda excepción contiene
  - Una copia de la "**pila**" de llamadas (secuencia de métodos que se llamaron para llegar al punto de fallo): `e.printStackTrace()`
  - El **mensaje** que se adjuntó a la excepción indicando su causa, en el momento en el que se creó: `e.getMessage()`
  - (Opcional), las **causas** de esta excepción, que serán a su vez excepciones con pila, mensaje y causas: `e.getCause()`
- Es mejor usar o extender tipos existentes (si son apropiados) en lugar de crear excepciones nuevas.

Iván Martínez Ortiz / Manuel Freire Morán

¿Qué hacer con una excepción?

- Dentro de un bloque `catch`, puedes
  - Manejarla

```
catch (IllegalArgumentException iae) {
    System.err.println("Error creando círculo: " + iae);
    iae.printStackTrace(); // muestra pila
    System.exit(0); // sale del programa (no recomendable!)
}
```

  - Decorarla

```
catch (IllegalArgumentException iae) {
    throw new IllegalArgumentException(
        "Creando círculo", iae); // cita iae como 'causa'
}
```

  - Propagarla (usando `throws`, si es de tipo *checked*): pasas el problema a quien te llamó, sin decorarla

```
public static void delete(File f) throws IOException {
    if (f.isDirectory()) { ... }
    f.delete(); // podría decorar una posible excepción, para dar más contexto
}

// se producirá una FileNotFoundException, delegada por el método delete()
delete(new File("fichero_inexistente.txt"));
```

Iván Martínez Ortiz / Manuel Freire Morán

Lanzando excepciones

- Basta con hacer un `throw` de una nueva instancia de la excepción elegida.
- El flujo del programa acabará ahí: un `throw` equivale a un `return` (si no está dentro de un `catch`).

```
public static void main(String[] args) {
   if (args.length < 2) {
      throw new IllegalArgumentException(
         "esperaba más argumentos");
   }
   // sólo llego si args.length >= 2
   System.out.println("El primer argumento es " +
      args[0] + " y el segundo, " + args[1]);
}
```

Iván Martínez Ortiz / Manuel Freire Morán

Creación de Excepciones propias

- Debe ser una clase que tenga como antecesor a `java.lang.Exception` o a `java.lang.RuntimeException`
- Elegir `Exception` cuando queremos que el cliente **gestione el error**
  - El objetivo habitual es reintentar la operación cambiando el contexto.
    - E.g. Escritura de fichero y no tienes permisos para escribir
- Elegir `RuntimeException` cuando se quiera indicar **un error de programación**
  - Violación del contrato del objeto (parámetros incorrectos, estado inconsistente)

```
public class MiExcepcion extends RuntimeException {
  public MiExcepcion(String message) {
    super(message);
  }
  public MiExcepcion(String message, Throwable cause) {
    super(message, cause);
  }
}
```

Iván Martínez Ortiz / Manuel Freire Morán

Algunas excepciones comunes

- java.lang.ClassCastException
  - Se ha realizado un cast (e.g. (MiClase)o) ilegal
- java.lang.IllegalArgumentException
  - Lanzada cuando un parámetro de un método no cumple los requisitos especificados en el contrato/documentación de la clase
- java.lang.NullPointerException
  - Lanzada cuando intentamos invocar un método de una variable tipo objeto que no está inicializada.
  - Lanzada cuando pasamos un argumento `null` a un parámetro de un método que no puede manejar nulos.
- java.lang.IndexOutOfBoundsException
  - Lanzada cuando accedemos a una colección u objeto que tiene una propiedad indexada por un índice y el argumento está fuera de rango.
- java.lang.IllegalStateException
  - Lanzada cuando el método invocado no puede ejecutarse en el estado actual del objeto. En otras condiciones, el método podría ejecutarse con normalidad.
- java.lang.UnsupportedOperationException
  - Lanzada cuando el método invocado no implementa el servicio esperado
- java.lang.ClassNotFoundException
  - Lanzada cuando la JVM no encuentra la clase que estamos utilizando en nuestra aplicación.

Iván Martínez Ortiz / Manuel Freire Morán

Logging: más allá del `System.out.println()`

- El uso indiscriminado de los `System.out` tiene problemas:
  - cuando dejan de ser necesarios, **es difícil desactivarlos** todos (hay que ir buscándolos y comentándolos uno a uno).
  - **producen** "**ruido**". En cada momento te interesan unos distintos; y los que no te interesan hacen más difícil leer el resto.
  - hacen que el programa vaya **más lento**; en un programa de cálculo intensivo, mostrar todos los pasos intermedios puede ser fácilmente la parte más lenta.

- Una alternativa es usar "logging" (trazas).
  - Hay varias librerías externas para ello: `log4j`, `slf4j`, `commons-logging`, ...
  - Lo más cómodo es usar la que viene con el propio Java en el paquete `java.util.logging`.

- Las trazas se pueden
  - Activar y desactivar por categorías enteras ("sólo las importantes")
  - Activar y desactivar por módulos enteros ("sólo los accesos a BD")
  - Redirigir a fichero, red, correo electrónico, etcétera.
    (pero por defecto funcionan igual que `System.out.println()`)

Iván Martínez Ortiz / Manuel Freire Morán

Java.util.logging

- En la cada clase en la que quieras generar trazas,

```
public static Logger log = Logger.getLogger("prefijo");
```

(el prefijo es algo que se mostrará junto con cada mensaje)

- Cuando tengas cosas que mostrar, usa

```
log.fine(texto) // para cosas detalladas
log.info(texto) // para informacion general
log.warn(texto) // para errores
// muestra la excepcion 'ex' y un texto como un 'warn'
log.log(Level.WARN, texto, ex)
```

Muestra la excepción con todo detalle: equivale a
- `ex.printStackTrace()`
- `System.out.println(ex.getMessage())`
- y *lo mismo para cada* `ex.getCause()` *encadenada*

muy recomendable

Iván Martínez Ortiz / Manuel Freire Morán

Manipulación de Cadenas

- La clase String permite hacer muchas de las operaciones típicas:
  - Longitud de una cadena: `s.length()`
  - Subcadenas: s.substring(inicio), `s.substring(`*`inicio, final`*`)`
  - Búsqueda textual de una subcadena: `s.indexOf(`*`texto`*`)`, `s.contains(`*`texto`*`)`, `s.lastIndexOf(`*`texto`*`)`, `s.startsWith(`*`texto`*`)`, `s.endsWith(`*`texto`*`)`, …
  - Ordenación sencilla (basada en localización): `s.compareTo(`*`texto`*`)`
  - Concatenación simple: `s + `*`texto`*
  - Rotura en subcadenas: `s.split(`*`expresión`*`)`

Iván Martínez Ortiz / Manuel Freire Morán

Para construir cadenas, `StringBuilder`

- Los `String` son inmutables. No puedes modificarlas, sólo crear copias modificadas.
- `StringBuilder` sí es mutable; construye tu cadena con él, y luego (sólo al final), pásalo a String, usando toString().

```
public static String concatenaMal(String[] cadenas) {
    String r = "";
    for (String c : cadenas) { r += c; } // feo !!
    return r;
}
public static String concatenaBien(String[] cadenas) {
    StringBuilder sb = new StringBuilder();
    for (String c : cadenas) { sb.append(c); }
    return sb.toString();
}
```

Iván Martínez Ortiz / Manuel Freire Morán

Convirtiendo a String: `toString()`

- Object contiene un método público de instancia `toString()`. Todos los objetos lo heredan.
- La implementación por defecto devuelve clase@puntero (ejemplo: `java.lang.Object@1cc2ea3f`)
- Sobreescribe `toString()`, y podrás usar el operador + para mostrar tus objetos

```
public class Persona {
    // …
    @Override // anotación opcional
    public String toString() {
        return apellidos + ", " + nombre + " (" + edad + " años)";
    }
}
// asumimos p persona
System.out.println("Alumno: "+p); // "Alumno: López, Paco (22 años)"
```

Iván Martínez Ortiz / Manuel Freire Morán

Expresiones regulares

- Forma compacta de representar patrones de cadenas. Disponible en `java.util.regex`
- Sintaxis en el Javadoc de `Pattern`. Ejemplos:
  - un teléfono de nueve dígitos tomados 3+2+2+2:
    `"\\d{3}-\\d{2}-\\d{2}-\\d{2}"`
  - una URL:
    `"^(https?|ftp|file)://[-a-zA-Z0-9+&@#/%?=~_|!:,.;]*[-a-zA-Z0-9+&@#%~_|]"`
  - una dirección IP, con puerto opcional:
    `"(\\d{3}).(\\d{3}).(\\d{3}).(\\d{3})(:([0-9]{1,5}))?"`

```
Pattern p = Pattern.compile("\\d{3}-\\d{2}-\\d{2}-\\d{2}");
String textoConTelefonos =
    "el 914-55-55-55 ha llamado al 653-55-55-55 20 veces en 2012";
Matcher m = p.matcher(textoConTelefonos);
while (m.find()) {
    System.out.println(m.group(0)); // muestra telefonos encontrados
}
```

Iván Martínez Ortiz / Manuel Freire Morán

Expresiones regulares: grupos de captura

- Usando paréntesis, es posible definir "grupos de captura", que permiten extraer fragmentos

```
public class DireccionIpV4 {
    public DireccionIP(int b1, int b2, int b3, int b4, int p) {…}
    // …
}

ArrayList<DireccionIpV4> dirs = new ArrayList<DireccionIpV4>();
Pattern dirIpConPuerto = Pattern.compile(
    "(\\d{3}).(\\d{3}).(\\d{3}).(\\d{3})(:([0-9]{1,5}))?");
String textoConIPs =
    "peticion del 150.244.56.240 al 150.244.56.32:8080, en t=50.122";
Matcher m = p.matcher(textoConIPs);
while (m.find()) {
    dirs.add(new DireccionIP(
        Integer.parseInt(m.group(1)), Integer.parseInt(m.group(2)),
        Integer.parseInt(m.group(3)), Integer.parseInt(m.group(4)),
        m.groupCount() > 4 ? m.group(6) : -1));
}
```

Iván Martínez Ortiz / Manuel Freire Morán

Más allá de expresiones regulares

- Las expresiones regulares son muy poderosas
  El código equivalente a una regex compleja sería largo, complejo, y difícil de modificar
- No son apropiadas para todos los casos:
  - "Determina si estos paréntesis están bien anidados"
  - "Verifica si este XML es válido"
  - "Comprueba que todos los números proporcionados son primos"
  - …
- Si necesitas algo más potente,
  - usa *parsers* existentes (por ejemplo, para XML)
  - escribe el tuyo (para casos muy sencillos: por ejemplo, anidación de paréntesis)
  - usa librerías para escribir parsers (por ejemplo, *antlr*)

Iván Martínez Ortiz / Manuel Freire Morán

Manipulación de cadenas: rompiendo texto

- Puedes romper un bloque de texto usado
  - *texto*`.split(`*expresion*`)`: genera un array con los fragmentos. Bueno si hay pocos (caro, fácil de usar).
  - `StringTokenizer(`*texto*`,`*separador*`)`: permite iterar por fragmentos. Bueno si hay muchos (barato).
  - *expresion* + *matcher* + `m.find()`: lo mejor de ambos mundos, pero más código para escribir.

```
// con StringTokenizer
StringTokenizer st = new StringTokenizer(textoMuyLargo, "\n");
for (int i=0; st.hasMoreTokens(); i++) {
     System.out.println("Linea " + i + ": " + st.next());
}
// con split
String[] lineas = textoMuyLargo.split("[\n]+");
for (int i=0; i<lineas.length(); i++) {
     System.out.println("Linea " + i + ": " + lineas[i]);
}
```

Iván Martínez Ortiz / Manuel Freire Morán

Limpiando texto

- Eliminando espaciado inicial y final:

```
limpio = texto.trim();
// "   ejemplo\n" -> "ejemplo"
```

- Pasando a minúsculas

```
limpio = texto.toLowerCase();
// "EsTrAfAlArIo" -> "estrafalario"
```

- Eliminando ciertos caracteres

```
limpio = texto.replaceAll("[^a-zA-Z0-9]+","_");
// "Los 40:\n\t¡qué tiempos!" -> "Los_40_qu_tiempos_"
```

- Recuerda: *las cadenas son inmutables*
  si no asignas el resultado a una variable, lo pierdes

Iván Martínez Ortiz / Manuel Freire Morán

Fechas y calendarios

- `java.util.Date` para almacenar fechas+horas.
- `java.util.GregorianCalendar` para manipularlas (maneja zonas horarias, etcétera).
- `java.text.SimpleDateFormat` para entrada/salida de fechas.

```
SimpleDateFormat formato =
    new SimpleDateFormat("yyyy.MM.dd@kk:mm:ss.SSS");
ParsePosition pp = new ParsePosition(0);
Date fecha = formato.parse("2012.09.25@12:50:14.900", pp);
System.out.println("fecha: " + formato.format(fecha));
GregorianCalendar gc = new GregorianCalendar();
gc.setTime(fecha);
gc.add(GregorianCalendar.DATE, 20);
System.out.println("Dentro de 20 días será " + gc);
Date fechaEn20 = gc.getTime();
System.out.println("fecha: " + formato.format(fechaEn20));
```

- Para tener un mayor control y evitar problemas es mejor utilizar la librería Joda Time

Iván Martínez Ortiz / Manuel Freire Morán

Cronometrando tiempos

- `System.nanoTime()` devuelve "un" tiempo en nanosegundos
  (NOTA: el valor exacto no es útil de forma aislada, pero puede representar nanosegundos desde que se arrancó el PC)
- Se pueden cronometrar intervalos usando dos muestras de `System.nanoTime()`.
- También es posible usar `System.currentTimeMillis()`, pero es bastante menos preciso
  (aunque sí devuelve un valor con significado: ms desde el 1 de enero de 1970)

```
long t0 = System.nanoTime();
// ... operación larga
long tiempoTranscurrido = System.nanoTime() - t0;
System.out.println("La operación ha tardado " +
    (tiempoTranscurrido / 1000000) + " ms");
```

Iván Martínez Ortiz / Manuel Freire Morán

Formatos de números

- Es posible leer muchos números simples usando `Tipo.valueOf(texto);`
- Pero sólo puedes controlar la salida si usas un **formato**. El mismo formato permite lectura y escritura.
- Similar a `SimpleDateFormat` en el uso de un patrón y los `ParsePosition` para indicar posición (en lectura): método `parse()`
- Al igual que los `SimpleDateFormat`, tienen en cuenta *localización* (y usa por defecto la del sistema)

```
DecimalFormat dfLocal = new DecimalFormat("0.000");
System.out.println(dfLocal.format(1000));
System.out.println(dfLocal.format(0));
System.out.println(dfLocal.format(Math.PI));
DecimalFormat dfIngles = new DecimalFormat("0.000",
        DecimalFormatSymbols.getInstance(Locale.US));
System.out.println(dfIngles.format(Math.PI));
```

```
1000,000
0,000
3,142
3.142
```

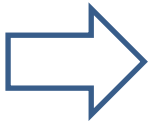
Iván Martínez Ortiz / Manuel Freire Morán

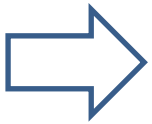
Números grandes y precisión

- Es imposible meter infinitos valores en un número de bits finitos (`double` y `long`: 64bits)
  - Los enteros tienen desbordamiento
  - Los flotantes, pérdida de precisión

```
double d=0;
for (int i=0; i<10; i++) {
    d+=.1;
    System.out.println(d);
}

// n = -(1<<31) = Integer.MIN_VALUE;
int n = -2147483648;
System.out.println(
    n + " - 1 = " + (n-1));
```





```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999

-2147483648 - 1 =
2147483647
```

Iván Martínez Ortiz / Manuel Freire Morán

Precisión infinita: `BigInteger` y `BigDecimal`

- Disponibles en `java.math`
- Pueden usar formatos igual que sus primos de tamaño fijo.
- Más lentos que `long` y `double` (*pero nunca pierden precisión*)

```
BigDecimal cantidad = new BigDecimal(
    "100000000000000.00"); // 100 000 millones
NumberFormat f = NumberFormat.getCurrencyInstance();
System.out.println(f.format(cantidad));
BigDecimal incrementada = cantidad.add(
    new BigDecimal("0.01"));
System.out.println(f.format(incrementada));
System.out.println(f.format(100000000000000.00 + .01));
```

```
100.000.000.000.000,00 €
100.000.000.000.000,01 €
100.000.000.000.000,02 €
```

Iván Martínez Ortiz / Manuel Freire Morán

Collections

- Contenedores estándares con
  - Secuencias de elementos
    (con acceso aleatorio y memoria contigua,
    o acceso secuencial y fácilmente insertables)
  - Conjuntos
    (ordenados y desordenados; no aceptan duplicados)
  - Mapas
    (conjunto de *claves* con un *valor* por clave)
- Interfaz común: si c es una `Collection<E>`…
  - `c.size()`: su tamaño (en elementos)
  - `c.add(E e)` / `c.addAll(Collection<E> es)`: añade elementos
  - `c.remove(E e)` / `c.removeAll(Collection<E> es)`: elimina
  - `c.clear()`: vacía la colección
  - `c.contains(E e)`: devuelve 'true' si e está en c
  - `c.toArray()` / `c.toArray(E[] es)`:
    devuelve una copia en un array (que puede estar ya reservado)

Iván Martínez Ortiz / Manuel Freire Morán

Tipos genéricos

- Desde la JDK 1.5 (antes se usaba Object en todos los contenedores)
- Permiten especificar los tipos que se van a usar en un contenedor (u otra clase).
- Una vez declarados, los 'argumentos de tipo' se usan con normalidad

```
public class Pareja<A,B> { // A y B son 'parámetros de tipo'
    private A a;
    private B b;
    public Pareja(A a, B b) {this.a = a; this.b = b;}
    public A getPrimero() { return a; }
    public B getSegundo() { return b; }
}
```

- Los parámetros de tipo tienen que ser *clases* (`int` no vale; `Integer`, sí)
- Uso: los 'argumentos de tipo' *forman parte* del tipo al que cualifican

```
Pareja<String, Integer> duracionMes
    = new Pareja<String, Integer>("septiembre", 30);
String nombreMes = duracion.getPrimero(); // "septiembre"
int diasMes = duracion.getSegundo(); // 30
String error = duracion.getSegundo(); // no compila
```

Iván Martínez Ortiz / Manuel Freire Morán

Iteradores e Iterables

- `Collection<E>` implementa `Iterable<E>`, que proporciona el método `public Iterator<E> iterator();`

- `Iterator<E>` proporciona
```
public boolean hasNext();
public E next();
public void remove();
```

- Dos formas de recorrer todos los elementos de una `Collection<Integer> c` usando iteradores:

```
// usa el iterador directamente
Iterator<Integer> it = c.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

recomendado,
*¡y además **funciona igual para arrays**!*

```
// usa un "bucle for para Iterables"
for (int i : c) {
    System.out.println(i);
}
```

Iván Martínez Ortiz / Manuel Freire Morán

El método equals()

- Los objetos de una secuencia, o de un conjunto no-ordenado, se comparan usando `equals()`
- Este método viene definido desde la clase Object (que lo implementa, por defecto, como '=='). *Sobreescribe este método para tus propias clases.*
- El contrato es:
  - `a.equals(b) == true` si 'a' y 'b' contienen *lo mismo*
  - `a.equals(b) == false` en caso contrario (incluyendo cuando `b` es `null`, o es de otro tipo)
- Eclipse puede generar una versión de equals() que compara campo a campo; *no es correcta* para todos los casos, pero puede ser útil como plantilla.

Iván Martínez Ortiz / Manuel Freire Morán

List: listas secuenciales

- Implementan la interfaz List
- Mantienen el orden de sus elementos
- Son lentas para buscar elementos (necesitan mirar en cada uno para ver si es el elemento buscado)
- Permiten el uso de índices con **`get`**/**`set`**/**`indexOf`**

```
List<String> lista =
    new ArrayList<String>();
    // new LinkedList<String>() también valdría

lista.add("hola");
lista.add("mundo");
lista.get(0); // "hola"
lista.get(1); // "mundo"
lista.set(0, "Hola"); // substituye el elemento en 0 por otro
lista.indexOf("mundo"); // compara con equals(), devuelve '1'
lista.indexOf("manzana"); // devuelve -1: no encontrado
```

Iván Martínez Ortiz / Manuel Freire Morán

ArrayList (simples) vs LinkedList (enlazadas)

- `ArrayList` almacena sus elementos en un array
  - `get(int i)` / `set(int i, E e)` es muy rápido
  - `add(int i, E e)` / `remove(int i)` son lentos (casi tanto como una búsqueda)
  - Es preferible, en la mayoría de los casos, usar `ArrayList` en lugar de un array normal.

- `LinkedList` usa una *lista enlazada*
  - todas las operaciones con índices son lentas (casi tanto como una búsqueda)
  - dispone de `ListIterator<E> listIterator()`, con el cual se pueden hacer operaciones de inserción en cualquier parte de la lista muy rápidamente
  - se usa menos frecuentemente que `ArrayList`

Iván Martínez Ortiz / Manuel Freire Morán

Set: conjuntos

- No mantienen el orden inicial de los elementos
- No permiten duplicados
- Inserción y eliminación de elementos algo más lenta que en listas
- Búsqueda muy rápida; útiles para hacer intersecciones o encontrar duplicados
- Dos tipos:
  - ordenados (`TreeSet<E>`): requieren un orden
  - desordenados (`HashSet<E>`): usan `hashCode()`
- Velocidad similar; `HashSet<E>` puede ser algo más rápido en ciertos casos.

Iván Martínez Ortiz / Manuel Freire Morán

orden: la interfaz `Comparable`

- Los conjuntos ordenados (`TreeSet<E>`) requieren que sus elementos sean comparables entre sí
  - Pueden usar un `Comparator<E>` externo, ó
  - Los elementos deben implementar `Comparable<E>`, que obliga a disponer de `public int compareTo(E e)`

```
public class Persona implements Comparable<Persona> {
    private String nombre;
    private String apellidos;
    // ...
    @Override
    public int compareTo(Persona p) {
        int c = apellidos.compareTo(p.apellidos);
        if (c != 0) {
            return c;
        }
        return  nombre.compareTo(p.nombre);
    }
}
```

Iván Martínez Ortiz / Manuel Freire Morán

el método hashCode()

- Al igual que `equals()`, se hereda de `Object`
- Debe devolver un entero
  1. que sea siempre *igual* para el mismo objeto:
     si `a.equals(b)` entonces `a.hashCode() == b.hashCode()`
  2. que, idealmente (pero de forma opcional), sea *muy distinto* para cualesquiera dos objetos que no sean **`equals()`**
- Si lo vas a usar, necesitarás implementar tu propia versión, *sobreescribiendo* la de `Object`.
- Cuanto mejor cumpla la condición 2., más eficientes serán los `HashSet<E>`.

```
public static class Persona implements Comparable<Persona> {
    private String nombre;
    private String apellidos;
    // ...
    @Override
    public int hashCode() {
        return  nombre.hashCode() + apellidos.hashCode();
    }
}
```

Iván Martínez Ortiz / Manuel Freire Morán

TreeSet (ordenados) vs HashSet (desordenados)

- `TreeSet<E>` usa un árbol de búsqueda ordenado
  - inserta, elimina y busca en tiempo ~log(size())
  - requiere un comparador para poder comparar los elementos entre sí.
  - permite extraer los elementos en orden, encontrar máximos y mínimos, encontrar todos los que son mayores o menores que uno dado, etcétera.
- `HashSet<E>` usa una tabla hash
  - inserta, elimina y busca en tiempo constante
    Pero sólo si tu `hashCode()` no devuelve lo mismo para demasiadas parejas de elementos distintos; si *siempre* devuelve lo mismo para todos, funcionará igual de lento que una lista.
  - requiere un `hashCode()` correcto para poder desordenar bien los elementos.
  - no garantiza el orden en el que se van a extraer los elementos; cada ejecución del programa puede devolver un orden distinto

Iván Martínez Ortiz / Manuel Freire Morán

Mapas y entradas

- Los mapas permiten asociar elementos *clave* con elementos *valor*.
  - Cada clave sólo puede estar asociada con un valor.
  - Se pueden ver como un conjunto de parejas clave-valor (`Map.Entry<K,V>`, accesible mediante `entrySet()`).
- Proporcionan búsqueda rápida por clave, y las operaciones
  - `V put(K clave, V valor)`
    inserta el valor valor asociado a la clave clave; devuelve el valor anterior asociado a esa clave, si lo hubiera. Rápida.
  - `V get(K clave)`
    devuelve el valor asociado a esa clave, o null si no se ha insertado ningún valor para esa clave. Rápida
  - `boolean containsKey(K clave)`
    devuelve 'true' si existe alguna entrada con esa clave. Rápida.
  - `boolean containsValue(V valor)`
    devuelve 'true' si existe ese valor para alguna clave. Operación lenta (requiere mirar todos los valores)
  - `Set<K> keySet()`
    devuelve el conjunto de claves que tienen valores asociados en este mapa
  - `Set<Map.Entry<K,V>> entrySet()`
    devuelve el conjunto de parejas clave-valor que componen este mapa

Iván Martínez Ortiz / Manuel Freire Morán

Ejemplos de uso de mapas: contando palabras

```
public static Map<String, Integer> cuentaDuplicados(
    String[] palabras) {
                            // ó HashMap<String, Integer>
    Map<String, Integer> m = new TreeMap<String, Integer>();
    for (String p : palabras) {
        if ( ! m.containsKey(p)) {
            m.put(p, 1);
        } else {
            m.put(p, m.get(p) + 1);
        }
    }
    return m;
}
```

```
String texto = "En un lugar de la mancha, de cuyo nombre...";
Map<String, Integer> duplicados =
    cuentaDuplicados(texto.split("[ ,.]+"));
for (String p : duplicados.keySet()) {
    System.out.println("'" + p +"': " +
        duplicados.get(p) + " veces");
}
```

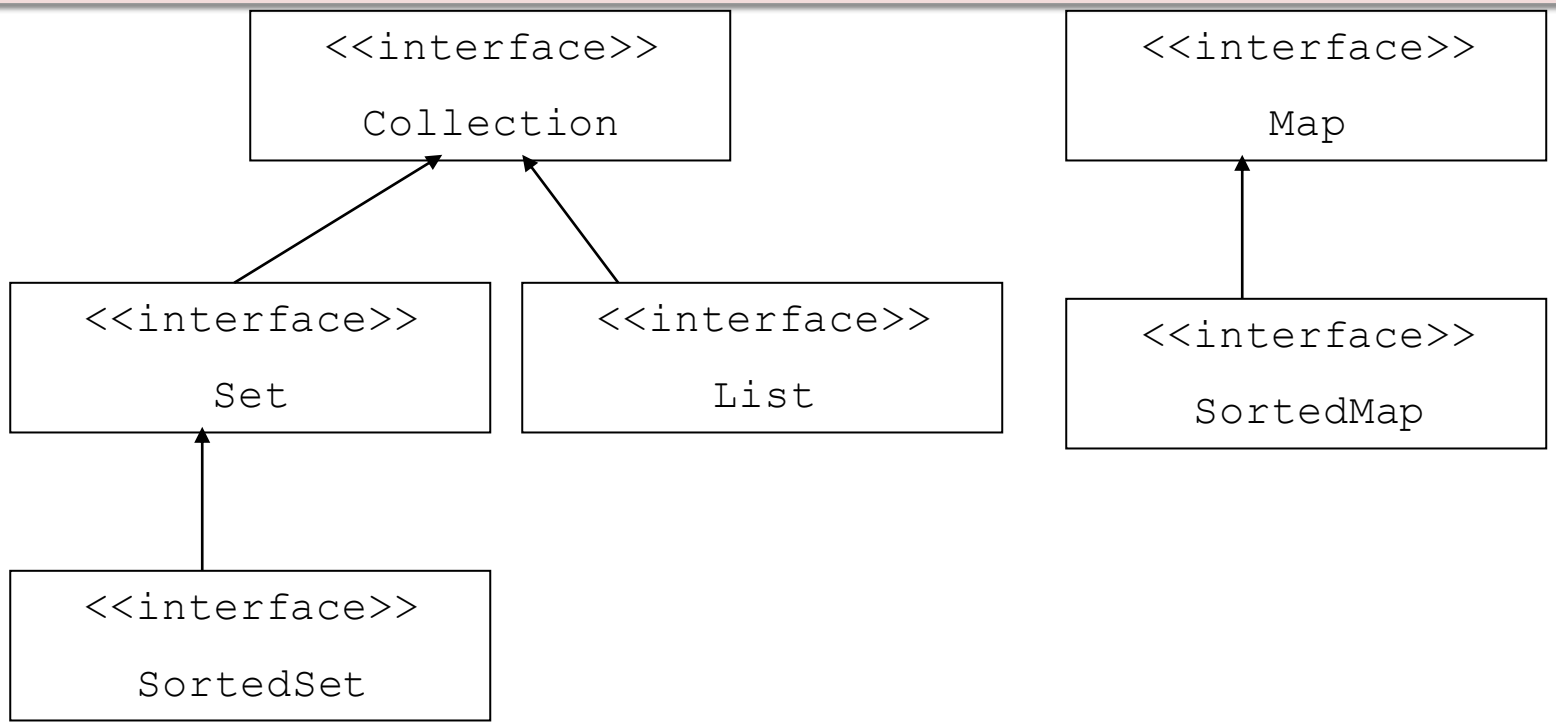
Iván Martínez Ortiz / Manuel Freire Morán

Mapas ordenados y desordenados

- Al igual los `HashSet<E>` y `TreeSet<E>`, los mapas vienen en dos sabores
- `HashMap<K, V>` usa, internamente, `HashSet<K>`
  - desordena sus elementos
  - requiere un buen `hashCode()` en `K`
  - tiene la misma eficiencia que un `HashSet<K>`
- `TreeMap<K, V>` usa, internamente, `TreeSet<K>`
  - ordena sus elementos de menor a mayor
  - requiere un comparador para `K`
  - tiene la misma eficiencia que un `TreeSet<K>`

Iván Martínez Ortiz / Manuel Freire Morán

Jerarquía de Colecciones



- Definen los contratos de las diferentes estructuras de datos
- Existen varias implementaciones por estructura
  - Cumplen el contrato de la interfaz pero con diferentes comportamientos y/o rendimientos.

Iván Martínez Ortiz / Manuel Freire Morán

Colecciones e implementaciones

		Implementaciones			
		Tabla hash	Array dinámico	Arbol balanceado	Lista enlazada
Interfaces	Set	`HashSet`		`TreeSet`	
	List		`ArrayList`, `Vector` `Stack`		`LinkedList`
	Map	`HashMap`, `Hashtable`, `LinkedHashMap`		`TreeMap`	

Iván Martínez Ortiz / Manuel Freire Morán

Anotaciones

- Es posible anotar **clases** y **métodos** Java con **anotaciones**: texto de la forma `@Algo` ó `@Algo(atributos)`

- Las anotaciones del código pueden estar disponibles
  - sólo en los fuentes (`RetentionPolicy.SOURCE`)
  - también en los ficheros compilados (`RetentionPolicy.CLASS`)
  - también en tiempo de ejecución (`RetentionPolicy.RUNTIME`)

- Las anotaciones se pueden usar para añadir funcionalidad:
  - **Java** usa anotaciones `@Override` para garantizar que los métodos que deben sobreescribir otros están, efectivamente, sobreescribiendo algo existente en su clase padre.
  - **Hibernate** usa anotaciones `@Entity` en clases para generar y mantener la estructura de una base de datos con una tabla por cada clase así marcada.
  - **Guice** usa anotaciones `@Inject` para hacer inyección de dependencias, facilitando la configuración y prueba de aplicaciones complejas.
  - **JUnit** usa anotaciones `@Test` para marcar los tests que debe ejecutar para probar un programa

Iván Martínez Ortiz / Manuel Freire Morán

Anotaciones: un ejemplo

```
// Prueba.java: define @Prueba
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Prueba {}
```

```
// Ejemplo.java: usa @Prueba
public class Ejemplo {
    @Prueba
    public static boolean m1() { }
    public static void m2() { }
    @Prueba public static int m3() {
        throw new
        RuntimeException("Boom!");
    }
}
```

NOTA

`EjecutaTests` usa *introspección*, que permite inspeccionar el contenido de un programa Java desde dentro del programa; es útil saber que existe, pero en la práctica se usa poco

```
// EjecutaTests.java: ejecuta métodos @Prueba
import java.lang.reflect.*;
public class EjecutaTests {
   public static void main(
     String[] args) throws Exception {

      int bien = 0, mal = 0;
      Class cp = Ejemplo.class;
      Class ca = Prueba.class;
      for (Method m : cp.getMethods()) {
         if (m.isAnnotationPresent(ca)) {
            try {
               m.invoke(null);
               bien ++;
            } catch (Throwable e) {
               System.out.println(
                  "Error en " + m + ": " +
                   e.getCause());
               mal ++;
            }
         }
      }
      System.out.println("Bien: " +
         bien + ", mal " + mal);
   }
}
```

Iván Martínez Ortiz / Manuel Freire Morán

Algunas Clases de Utilería

- java.lang.System
  - Acceso a la I/O básica: `System.{out | err | in }`
  - **Copia optimizada de arrays: `arrayCopy()`**
  - Acceso a las variables del sistema: `getProperties()`
    - Información del sistema y parámetros de configuración de la JVM.
  - Acceso a las variables de entorno: `getEnv()`
  - Gestión de librerías nativas: `loadLibrary()`
  - Finalización de la JVM con código de retorno: `exit(int)`
- java.lang.Runtime (interacción restringida con la JVM)
  - Ejecución de subprocesos: `exec()`
  - Recomendar recolección de basura: `gc()`
  - Memoria de la JVM: `freeMemory(), maxMemory(), totalMemory()`
- java.util.Collections (Utilerías para coleciones)
  - "Vista decorada" sólo lectura de las colecciones.
  - "Vista decorada" con soporte de concurrencia para las colecciones
  - Estructuras de datos e iteradores nulos (son objetos pero vacíos)
- java.util.Arrays
  - Utilerías para gestionar arrays
- java.util.Properties
  - Carga/guardado de preferencias simples
- java.util.Random / java.security.SecureRandom
  - Generadores de números aleatorios
- java.util.ResourceBundle / java.util.Locale
  - Herramientas para gestionar `i18n` y `l10n` en aplicaciones Java

Iván Martínez Ortiz / Manuel Freire Morán

Referencias

- Ejemplo de anotaciones: modificado de http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html
- Java (según Oracle):
  - docs.oracle.com/javase/tutorial/
  - docs.oracle.com/javase/7/docs/
  - docs.oracle.com/javase/specs/jls/se7/html/index.html

Iván Martínez Ortiz / Manuel Freire Morán