Desarrollo Android

Arkaitz Garro

Master en desarrollo de Apps Móviles

Java

para Android...

¿Existe un Java específico para Android?

NO

Un vistazo a Java

Introducción

Java: Introducción

Sintaxis similar a C y C++

Pero sin punteros, sin herencia múltiple, gestión automática de memoria...

Interpretado

Un programa (.java) se compila a código bytecode (.class)

El código bytecode lo interpreta una máquina virtual de Java.

Multiplataforma

Seguro...

Un vistazo a Java

POO

Java: Introducción

Encapsulación

Las estructuras de datos y los detalles de la implementación de una clase se hallan ocultos de otras clases del sistema

Control de acceso a variables y métodos

Herencia

Una clase (subclase) puede derivar de otra (superclase)

La subclase hereda todas los atributos y métodos de la superclase

Las subclase puede redefinir y/o añadir atributos y métodos

Java: Introducción

Polimorfismo

Es la capacidad de tener métodos con el mismo nombre (y argumentos) y diferente implementación

Una operación puede tener más de un método que la implementa

Sobrecarga

Un método puede tener distintas implementaciones con distintos tipos de argumentos

Java: Hello world!

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Java

Tipos de datos, variables y arrays

Tipos de datos

Java es un lenguaje fuertemente tipado

Tipos primitivos

byte -128 a 127

short –32,768 a 32,767

int –2,147,483,648 a 2,147,483,647

long –9,223,372,036,854,775,808 a 9,223,372,036,854,775,807

float, double, char, boolean

Variables

Declaración de una variable (atributo)

```
[access] [static] [final] type identifier [ = value];
```

Los nombres de variables distinguen entre mayúsculas y minúsculas.

El nombre de una variable puede ser cualquier secuencia de longitud ilimitada de letras y dígitos Unicode, que empiece con una letra, el símbolo del dólar «$» o el guión bajo «_».

Variables: visibilidad

```
class Scope {
  public static void main(String args[]) {
    int x; // known to all code within main
    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block
      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here
    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

Variables: conversión y casting

Conversión automática

Cuando los dos tipos con compatibles

Cuando el tipo de destino es mayor que el origen

Casting

Conversión de tipos incompatibles

```
(target-type) value|identifier;
```

```
int a;
byte b;
// ...
b = (byte) a;
```

Arrays

Declaración de arrays

```
type var-name[];
```

```
type var-name[] = new type[length];
```

```
type var-name[][] = new type[rows][cols];
```

```
int[] nums, nums2, nums3;
```

Java

Operadores

Operadores aritméticos

Operador	Resultado
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo
++	Incremento
+=	Suma y asignación
-=	Resta y asignación
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
- -	Decremento

Operadores relacionales

Operador	Resultado
==	Igual a
!=	No igual a
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

Operadores lógicos

Operador	Resultado
&	AND lógico
|	OR lógico
^	XOR lógico
||	Cortocircuito OR
&&	Cortocircuito AND
!	NOT
?:	Operador ternario

```
if (denom != 0 && num / denom > 10)
```

```
ratio = (denom == 0) ? 0 : num / denom;
```

Operadores de bits

Operador	Resultado
~	Operador NOT
&	Operador AND
|	Operador OR
^	Operador XOR
>>	Desplazamiento a la derecha
>>>	Desplazamiento a la derecha sin signo
<<	Desplazamiento a la izquierda
&=	Operador AND y asignación
|=	Operador NOT y asignación
^=	Operador XOR y asignación
>>=	Desplazamiento a la derecha y asignación
>>>=	Desplazamiento a la derecha sin signo y asignación
<<=	Desplazamiento a la izquierda y asignación

Java

Estructuras de control

IF

```
boolean dataAvailable;
// ...
if (dataAvailable)
    ProcessData();
else
    waitForMoreData();
```

```
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
} else {
    waitForMoreData();
    bytesAvailable = n;
}
```

```
if(month == 12 || month == 1 || month == 2)
    season = "Winter";
else if(month == 3 || month == 4 || month == 5)
    season = "Spring";
else if(month == 6 || month == 7 || month == 8)
    season = "Summer";
else if(month == 9 || month == 10 || month == 11)
    season = "Autumn";
else
    season = "Bogus Month";
```

SWITCH

```
switch(i) {
    case 0:
        System.out.println("i is zero.");
        break;
    case 1:
        System.out.println("i is one.");
        break;
    case 2:
        System.out.println("i is two.");
        break;
    case 3:
    case 4:
        System.out.println("i is 3 or more.");
        break;
    default:
        System.out.println("i is greater than 3.");
}
```

WHILE

```
while(n > 0) {
    System.out.println("tick " + n);
    n--;
}
```

```
do {
    System.out.println("tick " + n);
    n--;
} while(n > 0);
```

FOR

```
for(a=1, b=4; a<b; a++, b--) {
    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

Ejercicios

Java

Clases

Clases

Son una agrupación de datos o atributos, y métodos para operar sobre ellos

Definen un nuevo tipo de datos

```
[access] class ClassName {
    [access] [static] [final] type var1;
    [access] [static] [final] type var2;

    [access] type methodName1(parameter-list) {
        // body of method
    }

    // ...
    [access] type methodnameN(parameter-list) {
        // body of method
    }
}
```

Clases: atributos

Definen la estructura y los datos de los objetos de una determinada clase

```
[access] [static] [final] type var;
```

[access]: public, private, protected y package

[static]: variables propias de la clase, no de la instancia

[final]: constantes

Modifier	Class	Package	Subclass	World
`public`	✔	✔	✔	✔
`protected`	✔	✔	✔	✘
`no modifier*`	✔	✔	✘	✘
`private`	✔	✘	✘	✘

Clases: métodos

Definen las operaciones que se pueden realizar sobre una clase

```
[access] [static] [final] returnType methodName([args]){}
```

[access]: public, private, protected y package

[static]: llamada al método sin instanciar la clase

[final]: constantes, evita que sea sobreescrito

returnType: tipo de retorno

[args]: argumentos, separados por comas

Clases: métodos

Devolución de valores

En la declaración se debe especificar el tipo de valor devuelto

Si el método no devuelve nada, su tipo es **void**

El cuerpo del método deberá contener `return valor;`

```
class Punto {
  private int x;
  private int y;

  public double calcularDistanciaCentro() {
    double z;
    z = Math.sqrt((x*x)+(y*y));
    return z;
  }
}
```

Clases: instancias y uso

Instanciar un nuevo objeto

```
Punto p = new Punto();
```

Acceder a sus métodos

```
Double x = p.calcularDistanciCentro();

System.out.println(“Distancia: ” + x);
```

Clases: constructor

Método que se llama automáticamente cada vez que se crea un nuevo objeto

Reserva de memoria

Inicializar las variables de la clase

No tiene valor de retorno

Tiene el mismo nombre que la clase

Sobrecarga

Una clase puede tener varios constructores, que se diferencian por el tipo y el número de argumentos

Clases: constructor

Constructor sin argumentos

Inicializa tipos primitivos a sus valores por defecto

Inicializa objetos a **null**

```
Punto(int x) {this.x = x;}

Punto(int x, int y){this.y = y;}
```

Ejercicios

Clases: sobrecarga de métodos

Puede haber dos o más métodos que se llamen igual en la misma clase

Se tienen que diferenciar en los argumentos (número o tipo)

El valor de retorno no es suficiente para diferenciarlos

```
class Punto {
  public double test(){System.out.println(“Vacio.”);}
  public double test(int a){System.out.println(“a: ”+a);}
  public double test(int a, int b){System.out.println(“a y b: ”+a+” ”+b);}
  public double test(double a) {System.out.println(“Double a: ”+a);}
}
```

Clases: atributo y método estático

Los elementos (atributos y métodos) definidos como `static`, son independientes de las instancias de la clase

Un atributo **`static`** es una variable global de la clase

Una instancia no copia el valor del atributo, se comparte entre todas las instancias

```
class Punto {
   static int instancias;
}

System.out.println(“Número de instancias: “+Punto.instancias);
```

Clases: atributo y método estático

Métodos estáticos

Un método **`static`**, es un método global a la clase

Una instancia no copia el método estático

Una instancia no copia el valor del atributo, se comparte entre todas las instancias

Suelen utilizarse para para acceder a atributos estáticos

No pueden hacer uso de la referencia **`this`**

```
class Punto {
  static int instancias;
  static actualizarInstancias() {instancias++;}
}

System.out.println(“Número de instancias: “+Punto.instancias);
```

Clases: final

La palabra reservada final indica que el `valor` no se puede cambiar

Una clase final no puede tener clases hijas

Un método no puede ser sobrescrito en clases hijas

Una variable tiene que ser inicializada al declararse, y no puede cambiarse su valor

El identificador de una variable suele escribirse en mayúsculas

Java

Clases útiles

Clases útiles: String

Representa una cadena de caracteres

```
String s = new String();

char chars[] = {‘a’, ‘b’, ‘c’, ‘d’};
String s = new String(chars);

String s1 = “Hola!”;
String s2 = new String(s1);

String s3 = “Tengo ” + edad + “ años”;
```

Clases útiles: String

Extraer un carácter de una determinada posición

```
String s = “Hola!”;
System.out.println(s.charAt(0)); //’H’
```

Comparación de caracteres

`s.equals(“Texto”)`: compara el parámetro con el objeto

`s.equalsIgnoreCase(“Texto”)`: compara el parámetro con el objeto, ignorando mayúsculas y minúsculas

.equals no es lo mismo que ==

El operador == devuelve true si las dos variables referencian al mismo objeto

Clases útiles: String

Comparación de caracteres

```
String s = “Hola!”;
s.startsWith(“Hol”); //True
s.endsWith(“ola”); //False

s.compareTo(arg);
```

Búsqueda de caracteres

```
String s = “Hola!”;
s.indexOf(“ol”); //1
s.lastIndexOf(“la”); //2
```

Otras operaciones

`substring, concat, replace, toLowerCase, trim, valueOf...`

Clases útiles: Vector

Representa un array variable de objetos

Es una alternativa mucho más potente al uso de array

Dispone de métodos extra para el tratamiento de los elementos del array

```
Vector v = new Vector();
```

Clases útiles: Vector

Métodos

`size()`: número de elementos almacenados en el vector

`capacity()`: capacidad actual

`add(Object o)`: añade un nuevo elemento

`add(int pos, Object o)`: añade un nuevo elemento en una posición concreta, desplazando el resto

`firstElement()/lastElement()`: devuelve el primer/último elemento del vector

`get(int pos)`: devuelve el elemento en una determinada posición

Clases útiles: Vector

Métodos

`contains(Object o)`: comprueba si existe el elemento en el vector

`elements()`: devuelve un objeto de tipo Enumeration con todos los elementos del vector

`indexOf(Object o)`: devuelve la posición del objeto en el vector

`setElementAt(Object o, int pos)`: inserta un objeto en una determinada posición

`remove(int pos)`: elimina un objeto de una determinada posición

Clases útiles: Math

La clase Math contiene métodos que realizan operaciones matemáticas

`Math.sqrt(double a):` raíz cuadrada

`Math.pow(double a, double b):` a elevado a b

`Math.abs(double a):` valor absoluto

`Math.max(double a, double b):` valor máximo

`Math.min(double a, doble b):` valor mínimo

`Math.round(double a):` parte entera

...

Ejercicios

Java

Herencia y polimorfismo

Herencia: ¿qué es?

Consiste en la creación de nuevas clases a partir de otras ya existentes

Cuando una clase hereda de otra, contiene los atributos y métodos de la clase padre

La herencia permite

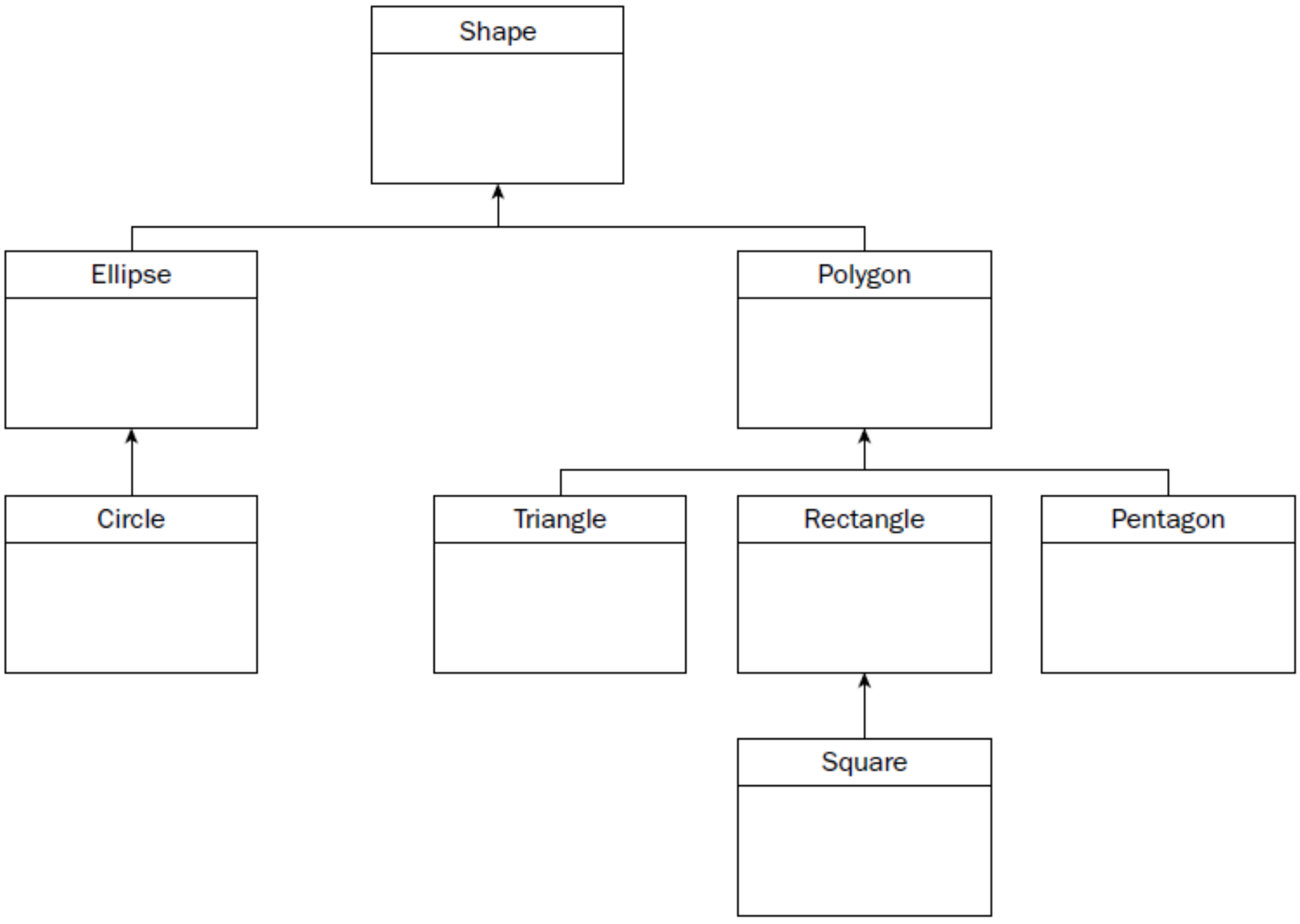
La reutilización de código

Añadir nuevos comportamientos a las clases heredadas

Redefinir comportamientos (métodos)

Crear una jerarquía de clases

Herencia: ¿qué es?



Herencia: ¿qué es?

Java no permite la herencia múltiple

La clase `Java.lang.Object` es la base de toda la jerarquía de objetos en Java

Si al definir una clase, no se especifica de qué clase deriva, por defecto hereda de **`Java.lang.Object`**

Esta clase proporciona métodos comunes a todas las clases:
`toString(), equals(), getClass(), clone()...`

Herencia: aspectos generales

```
class ClaseHija extends ClasePadre
```

Representa una relación “es un”

`extends` especifica que la clase hija hereda datos y métodos de la clase padre

Aunque hereda “todo”, la clase hija solo tiene acceso a los miembros de la clase padre, con modificadores `public`, `protected` o `package`

Herencia: super

`super` permite acceder tanto al constructor como a los métodos de la clase padre

Se puede llamar al constructor desde una subclase utilizando la palabra `super([args])` en la primera sentencia

Si no existe la llamada explícita, esta realiza de todas maneras

```
class A { A(){System.out.println(“Constructor de A”);}; }

class B extends A { B(){System.out.println(“Constructor de B”)}; }

class Principal {
  public static void main(String args[]) {
    B b = new B();
  }
}
```

Herencia: overriding

Un método se puede especializar, redefinir o sustituir en una subclase

El tipo de retorno, nombre, tipo y número de argumentos debe ser el mismo

Las excepciones lanzadas deben ser, como mucho, las declaradas en el método a redefinir

Los modificadores deben ser al menos, tan generales como los métodos de la superclase

Si el método de la superclase es `final`, no se puede sobrescribir

Herencia: ejemplo

```
class Figura {
  public int base;
  public int altura;

  public double calcularArea(){
    return base*altura;
  }
}

class Triangulo extends Figura {
  public double calcularArea() {
    return base*altura/2;
  }
}
```

Herencia: abstract

Una clase abstracta es aquella que no puede ser instanciada

Sirven para proporcionar un modelo que deben seguir las clases hijas y algunos métodos de utilidad general

```
[access] abstract class ClaseAbstracta {}
```

Los métodos en las clases abstractas

Pueden ser abstractos. Esto obliga a que la clase sea abstracta, y a implementar el método en las subclases

Métodos no abstractos

Constructores que pueden ser llamados con `super`

Ejercicios

Java

Interfaces

Interfaces: ¿qué son?

Es una colección de definiciones de métodos (sin implementación) y de valores estáticos y constantes

Define la interacción, no la conducta

Si se definen propiedades, por defecto son `static` y `final`

Una interfaz es análoga a una clase abstracta con todos sus métodos abstractos. Los métodos no pueden ser privados.

Las clases que implementan la interfaz, deben implementar todos los métodos definidos en la interfaz

Interfaces: ¿qué son?

Una clase puede heredar de una sola clase (`extends`), pero puede implementar varias interfaces (`implements`)

Una clase que implementa una interfaz puede tener método propios

Una interfaz NO se puede instanciar

Los métodos que se implementan de una interfaz deben ser públicos

Si se hereda el mismo método de varias interfaces, el método será único en la clase

Se permite la herencia entre interfaces

Herencia: ejemplo

```
[access] interface MiInterfaz {
  [access] returnType method1([args]);
  [access] returnType method2([args]);
}

[access]  class MiClase implements MiInterfaz {
  public returnType method1([args]) {}
  public returnType method2([args]) {}
  ...
}
```

Herencia: ejemplo

```
public interface Dibujable {
  public void setPosicion(double x, double y);
  public void dibujar(Graphics dw);
}

public class Circulo implements Dibujable {
  public void setPosicion(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public void dibujar(Graphics dw) {
    dw.paint();
  }

  ...
}
```

Ejercicios

Java

Excepciones

Excepciones: aspectos generales

Una excepción es un error en tiempo de ejecución

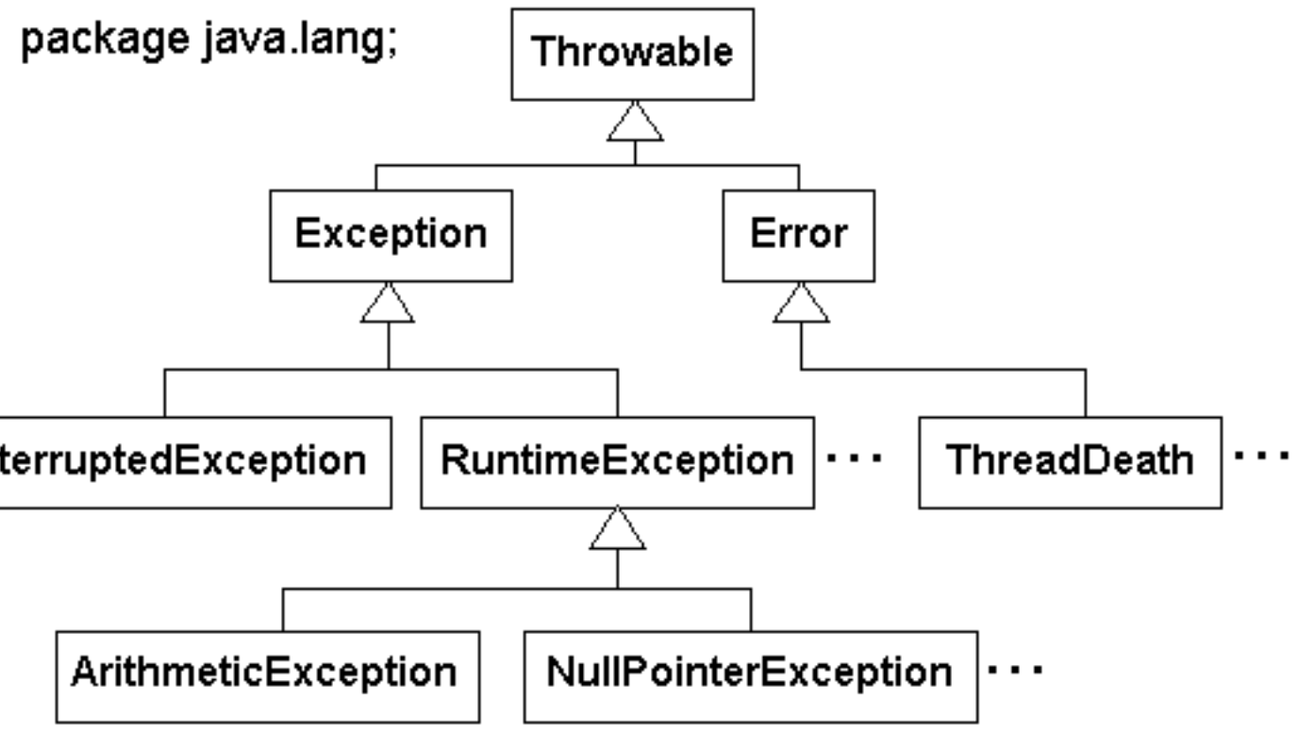
Tipos de excepción

Fatal: el programa debe finalizar (stack overflow)

Recuperable: tenemos la oportunidad de corregir el error

Las excepciones permiten escribir código para manejar el error y continuar con la ejecución del programa

Excepciones: aspectos generales



Excepciones: aspectos generales

Clase `Throwable`: superclase que engloba todas las excepciones

Clase `Error`: errores de compilación, del sistema o de la JVM normalmente irrecuperables

Clase `Exception`: define las excepciones que el programa debería tratar (`IOException`, `ArithmeticException`, etc...)

Por heredar de `Throwable`, pueden usar los métodos:

`getMessage()`: mensaje asociado a la excepción

`toString()`: descripción de la excepción

`printStackTrace()`: traza de la pila de llamadas

Excepciones: aspectos generales

Cuando ocurre un error

Se crea un objeto Exception de la clase adecuada con información sobre el error ocurrido.

Se lanza la excepción

Se propaga por la pila de métodos

Hasta que se captura en algún método

O se finaliza la pila de métodos

Excepciones: captura

Ciertos métodos del API de Java y algunos creados por el programador producen excepciones

Si se llama a estos métodos sin tenerlos en cuenta, se produce un error de compilación

Gestionar la excepción con las sentencias `try-catch-finally`

Relanzar la excepción hacia un método superior, declarando que este método también lanza dicha excepción

Es posible capturar la excepción, y no hacer nada con ella...

Excepciones: captura

El control de excepciones se realiza con bloques `try-catch-finally`

El código susceptible a errores debe estar acotado en el bloque `try`

Cuando se produce un error en el bloque `try`, se genera una excepción del tipo que se haya producido

Esta excepción puede ser recogida en el bloque catch.

Se pueden definir tantos bloques `catch` como sean necesarios.

El bloque `finally` es opcional, las sentencias de este bloque se ejecutan siempre

Excepciones: ejemplo

```
public void leerFichero() {
   try {
      Abrir el fichero;
      Leer el fichero en memoria;
      Cerrar el fichero;
   } catch(FalloApertura) {
      Hacer algo;
   } catch(FalloLectura) {
      Hacer algo;
   } catch(FalloCierre) {
      HacerAlgo;
   }
}
```

Excepciones: ejemplo

```
class Excepciones {
  public static void main(String args[]) {
    try {
      int a = args.length;
      int b = 10 / a;
      int c[] = {1};

      c[42] = 100;
    } catch(ArithmeticException e) {
      System.out.println(“División por 0.”);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.prinln(“Desbordamiento.”);
    }
  }
}
```

Excepciones: lanzamiento

Un método puede lanzar una o varias excepciones, que deben declararse mediante una cláusula throws

Si se lanzan varias excepciones, se separan por comas

```
public void calcular() throws ArithmeticException,
        ArrayIndexOutOfBoundsException {}
```

Ejercicios