Estructuras de Datos
Clase 1 - Introducción

Dr. Sergio A. Gómez
http://cs.uns.edu.ar/~sag

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Trivia

- Asignatura: Estructuras de datos
- Profesor: Dr. Sergio Alejandro Gómez
- Email: sag@cs.uns.edu.ar
- Horarios:
  - Teoría: Martes y jueves de 14 a 16 hs (Aula 8 Palihue)
  - Práctica: Martes y jueves de 16 a 18 hs (Aula 8 Palihue)
- Horario de consulta del profesor: A confirmar
- Horarios de consulta del asistente: A confirmar por el asistente

Bibliografía

- M.T. Goodrich & R. Tamassia. Data Structures and Algorithms in Java. Fourth Edition. John Wiley & Sons, 2006.
- Nota: El curso hará un énfasis importante en la lectura del libro.
- Adicionalmente se usarán apuntes de la cátedra o diapositivas en la práctica que se distribuirán oportunamente.
- Nota: Las diapositivas no son exhaustivas.

Motivaciones

- Estamos interesados en construir *buenas* estructuras de datos y *buenos* algoritmos.
- Estructura de datos = Manera sistemática de organizar y acceder datos.
- Algoritmo = Procedimiento paso a paso para realizar una tarea en una cantidad finita de tiempo.

Objetivos

- Continuación y profundización de los conceptos vistos en IPOO.
- Aprendizaje de:
  - Estructuras de datos (ED) fundamentales
  - Elección de una estructura de datos para un dominio determinado
  - Algoritmos asociados a las estructuras de datos
  - Evaluación de algoritmos (orden de tiempo de ejecución)

Factores de calidad del software

- Usaremos POO porque permite construir software de buena calidad.
- ¿Cómo evaluar la calidad del software?
  - Factores externos de calidad: Cualidad del software que puede ser detectada por un usuario (sin conocer el código fuente).
  - Factores internos de calidad: Cualidad del software que puede ser percibida por el programador (conociendo el código fuente).

Factores de calidad

- Correctitud: El programa respeta la especificación (para una entrada dada produce la salida esperada).
- Robustez: El programa es capaz de manejar entradas no contempladas por la especificación (se espera que se degrade "graciosamente" o "gracefully"). Ej.: Recibe un string cuando espera un entero. Factor crítico en aplicaciones industriales.
- Extensibilidad: Capacidad de agregarle funcionalidad al sotf.

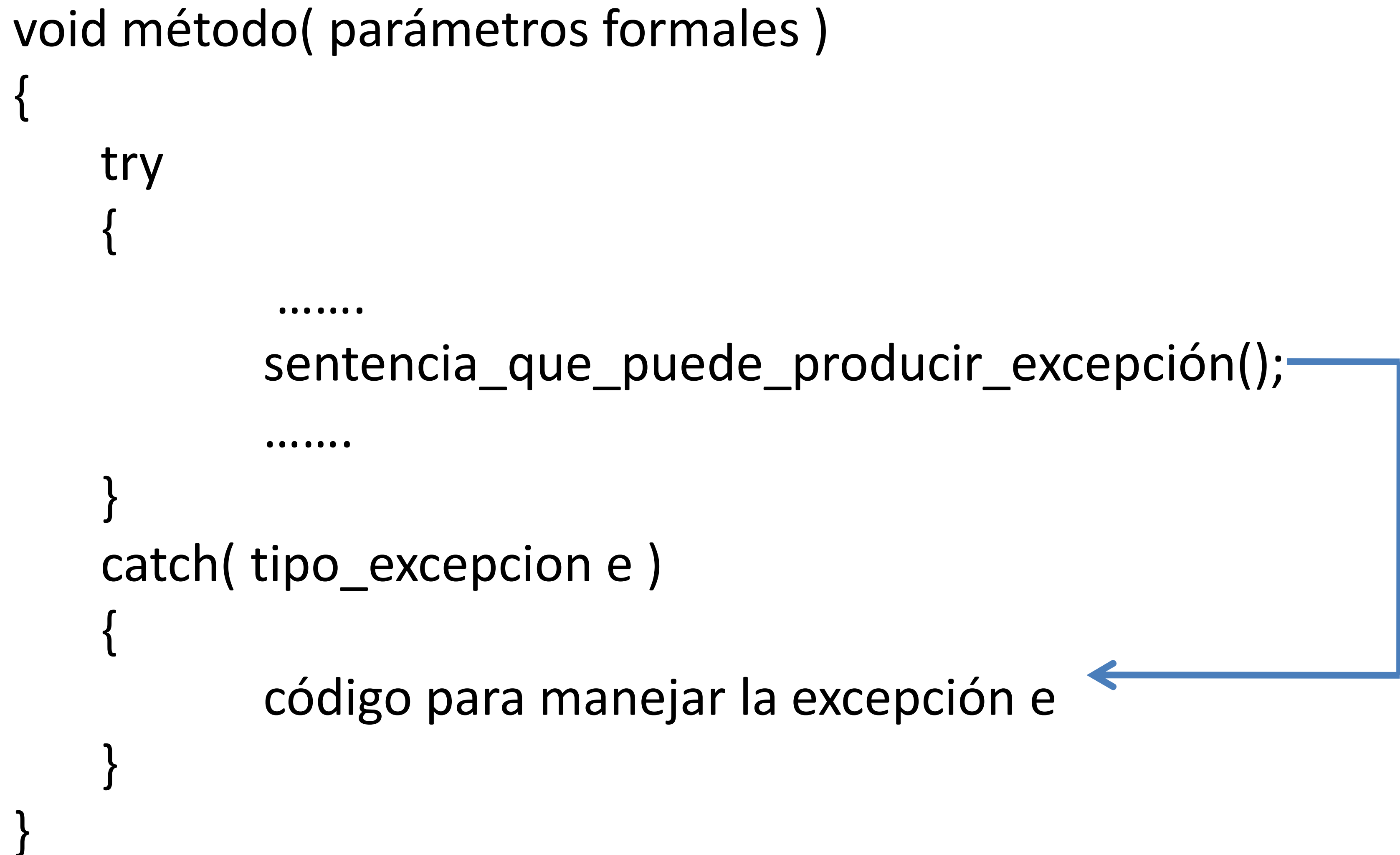
Factores de calidad

- Portabilidad: Habilidad del soft de correr con mínimo cambio en otro hard o sistema operativo. (por ello usamos Java)
- Reusabilidad: Capacidad de usar el código en la construcción de otros programas.
- Eficiencia: Capacidad del soft de requerir pocos recursos de cómputo (tiempo de CPU y memoria).
- Facilidad de uso: Capacidad del soft para ser usado por usuarios con distinta habilidad.

Robustez: Excepciones

- Las excepciones son eventos inesperados que ocurren durante la ejecución de un programa.
- Puede ser el resultado de una condición de error o una entrada incorrecta.
- En POO las excepciones son objetos “lanzados” por un código que detecta una condición inesperada.
- La excepción es “capturada” por otro código que repara la situación (o el programa termina con error en ejecución).

Excepciones: try-catch



Excepciones: Ejemplo

<u>Problema:</u> Leer un entero “pos” y mostrar la componente “pos” del arreglo “a”, el doble y el triple.

```
import java.util.Scanner;
import java.io.*;

public class EjExcepcion01
{
    public static void main( String [] args )
    {
        try {
            int [] a = { 1, 2, 3, 4, 5 };
            Scanner s = new Scanner( System.in );
            System.out.print( "Ingrese una posicion entre 0 y 4: " );
            int pos = s.nextInt();
            System.out.println( "El valor en " + pos + " es " + a[pos] );
            System.out.println( "El doble es " + 2 * a[pos] );
            System.out.println( "El triple es " + 3 * a[pos] );
        } catch( ArrayIndexOutOfBoundsException e ) {
            System.out.println( "Valor fuera de rango" );
        }
    }
}
```

```
>java EjExcepcion01
Ingrese una posicion entre 0 y 4: 40
Valor fuera de rango
```

Excepciones: try-catch-finally

```
try   {
        …….
}  catch( tipo_excepcion_1 e ) {
        código para manejar la excepción e
} catch( tipo_excepcion_2 e ) {
        código para manejar la excepción e
} finally {
      código que siempre se ejecuta
}
```

Excepciones: Ejemplo

```java
try {
    int [] a = { 1, 2, 3, 4, 5 };
    Scanner s = new Scanner( System.in );
    System.out.print( "Ingrese una posicion entre 0 y 4: " );
    int pos = s.nextInt();
    System.out.println( "El valor en " + pos + " es " + a[pos] );
    System.out.println( "El doble es " + 2 * a[pos] );
    System.out.println( "El triple es " + 3 * a[pos] );
} catch( ArrayIndexOutOfBoundsException e ) {
    System.out.println( "Valor fuera de rango" );
} catch( InputMismatchException e ) {
    System.out.println( "No ingreso un entero" );
} finally {
    System.out.println( "Adios!" );
}
```

```
>java EjExcepcion02
Ingrese una posicion entre 0 y 4: mama
No ingreso un entero
Adios!

>java EjExcepcion02
Ingrese una posicion entre 0 y 4: 2
El valor en 2 es 3
El doble es 6
El triple es 9
Adios!
```

Cómo crear nuestras excepciones

- Las excepciones son subclases de la clase Exception de Java.
- Los métodos deben especificar en su signatura las excepciones que pueden lanzar:

  tipo método(parámetros formales) throws clase_excepción

- Una excepción se lanza desde una sentencia throw:

  throw new clase_excepción(parámetros actuales);

- Ejemplo: Programar una clase persona. Una persona tiene un nombre y una edad. La edad debe ser un número positivo.
- Crearemos una excepción *PersonaException* para lanzar cuando el cliente ingresa una edad negativa en el constructor.

```java
// PersonaException.java
public class PersonaException extends Exception {
    public PersonaException( String msg ) {
        super( msg );
    }
}
```

```java
// Persona.java
public class Persona {
    private String nombre;
    private int edad;

    public Persona( String nombre, int edad ) throws PersonaException {
        this.nombre = nombre;
        if( edad <= 0 )
            throw new PersonaException( "Edad negativa: " + edad );
        this.edad = edad;
    }

    public String getNombre() { return nombre; }
    public int getEdad() { return edad; }
    public String toString() { return getNombre() + " " + getEdad(); }
}
```

```java
// EjException03.java
public class EjException03
{
    public static void main( String [] args )
    {
        try {
            Persona p = new Persona( "Juan", 30 );
            System.out.println( "p: " + p );
            Persona q = new Persona( "Tito", p.getEdad() - 35 );
            System.out.println( "q: " + q );
        } catch( PersonaException e ) {
            System.out.println( e.getMessage() );
        }
    }
}
```

Propagación de excepciones por la pila de registros de activación

```
public class EjExcepcion04
{
    private class MiExcepcion extends Exception
    { public MiExcepcion(String msg) {super(msg);} }

    private void w(String s) { System.out.println(s); }
    public void m1() throws MiExcepcion { w("m1"); m2(); w("m1-"); }
    public void m2() throws MiExcepcion { w("m2"); m3(); w("m2-"); }
    public void m3() throws MiExcepcion { w("m3"); m4(); w("m3-"); }
    public void m4() throws MiExcepcion {
        w("m4"); throw new MiExcepcion("Oooh!"); }

    public static void main(String [] args )
    {
        try {
            (new EjExcepcion04()).m1();
        } catch( MiExcepcion e ){
            System.out.println( e.getMessage() );
        }
    }
}
```

```
>java EjExcepcion04
m1
m2
m3
m4
Oooh!
```

Las excepciones no manejadas se propagan al llamante.

Notas

- Las excepciones como ArrayIndexOutOfBoundsException se llaman *unchecked* porque el programa compilará aunque el cliente no tengan un try-catch asociado. Esto ocurre con las RuntimeExceptions.
- Las excepciones como PersonaException se llaman *checked* porque el cliente no compilará si el código cliente no está protegido con el try-catch correspondiente.

Notas

- Las excepciones se estructuran en una jerarquía de herencia, al utilizar cláusulas catch múltiples, en el caso de una excepción éstas se comprueban en forma secuencial hasta que la el tipo de la excepción lanzada conforma al de la excepción declarada en el catch y recién en ese momento se ejecuta este último bloque catch.
- Entonces estructuras los catch de excepción más específica a más general.

Escribir:

```
try {
  // Código que puede lanzar una excepción
} catch( IOException e ) {
  // Código para procesar la excepción de entrada / salida
… } catch( Exception e ) {
 // Código para procesar un excepción general
…}
```

Y no:

```
try { ….
} catch( Exception e ) {
…
} catch( IOException e ) {  // Código inalcanzable ya que esta
// excepción es caputada siempre por el bloque catch anterior.
…}
```

Algunas excepciones Runtime definidas en java.lang

Runtime exception	Significado
ArithmeticException	Error de aritmética, e.g. división por cero
ArrayIndexOutOfBoundsException	Indice de arreglo fuera de límites
ClassCastException	Cast inválido
IndexOutOfBoundsException	Algún índice de array fuera de límites
NegativeArraySizeException	Array creado con tamaño negativo
NullPointerException	Uso incorrecto de una referencia nula
NumberFormatException	Conversión inválida de string en número
StringIndexOutOfBounds	Se quiso acceder a una posición inexistente de un string

Algunas excepciones unchecked de Runtime definidas en java.lang (informativa)

Runtime exception	Significado
ArithmeticException	Error de aritmética, e.g. división por cero
ArrayIndexOutOfBoundsException	Indice de arreglo fuera de límites
ClassCastException	Cast inválido
IndexOutOfBoundsException	Algún índice de array fuera de límites
NegativeArraySizeException	Array creado con tamaño negativo
NullPointerException	Uso incorrecto de una referencia nula
NumberFormatException	Conversión inválida de string en número
StringIndexOutOfBounds	Se quiso acceder a una posición inexistente de un string

Sobre el final del curso puede aparecer ClassNotFoundException.

Métodos útiles definidos por Throwable (superclase de Exception)

Método	Significado
String getMessage()	Retorna una descripción de la excepción
void printStackTrace()	Imprime a la consola la pila de registros de activación con nombre de método y número de línea en cada método donde se propagó la excepción hasta que fue capturada.
String toString()	Retorna una descripción de la excepción

Interfaces

- Una interfaz es una colección de declaraciones de métodos.

```
public interface nombre_interfaz
{
    public tipo método1( parámetros formales ) throws excepciones;
    ...
    public tipo métodon( parámetros formales ) throws excepciones;
}
```

```java
// Vendible.java
public interface Vendible {
    public String descripcion();
    public int precioLista();
}
```

```java
// Transportable.java
public interface Transportable {
    public int peso();
    public boolean peligroso();
}
```

Implementación de interfaces

- Una clase que implementa una interfaz debe implementar todos los métodos provistos por la interfaz.

```
public class mi_clase implements nombre_interfaz
{
    …. Lista de atributos …
    public mi_clase()  {  … código del constructor …  }
    public tipo método1( parámetros formales ) throws excepciones
    { … código del método … }
     ...
    public tipo métodon( parámetros formales ) throws excepciones
    { … código del método … }
}
```

```java
// Fotografia.java
public class Fotografia implements Vendible {
    private String descripcion;
    private int precio;
    private boolean color;

    public Fotografia( String desc, int p, boolean c )
    { descripcion = desc; precio = p; color = c; }

    // Implemento los dos métodos de la interfaz vendible:
    public String descripcion() { return descripcion; }
    public int precioLista() { return precio; }

    // Agrego los métodos que considere necesarios:
    public boolean color() { return color; }
}
```

Implementación de más de una interfaz

La clase *ItemEmpacado* implementa las interfaces *Vendible* y *Transportable*:

```
public class ItemEmpacado implements Vendible, Transportable {
    // Atributos:
    // Constructor:
    // Implementación de todos los métodos de ambas interfaces:
    // Implementación de métodos adicionales:
}
```

Herencia múltiple de interfaces

La interfaz *ItemAsegurable* extiende a las interfaces *Vendible* y *Transportable* y las extiende con el método *valorAsegurado*:

```
public interface ItemAsegurable extends Vendible, Transportable {
    public int valorAsegurado();
}
```

Ventajas del uso de interfaces

- Al usar una interfaz uno especifica "qué" debe hacer una clase pero no "cómo" debe hacerlo.
- Las interfaces no hacen suposiciones de "cómo" serán implementadas.
- Una vez que se define una interfaz, ésta puede ser implementada por cualquier número de clases.
- Implementar una interfaz requiere implementar todos los métodos definidos por la misma.
- Brindan una forma limitada y segura de herencia múltiple (una clase puede implementar más de una interfaz).

Ventajas del uso de interfaces

- Las interfaces están pensadas para soportar la resolución de métodos en tiempo de ejecución.
- Esto se puede lograr con clases abstractas también pero las mismas deberían estar más y más arriba en la jerarquía oscurenciéndola (este problema se evita con las interfaces ya que dos clases no relacionadas pueden implementar la misma interfaz).

Reglas para asignar en forma válida: a = b;

Supongamos que *a* es de tipo *U* y *b* de tipo *T*. Es válido asignar cuando:

- “a” y “b” son del mismo tipo de clase o interfaz (i.e. *T* = *U*).
- Cuando *T* es más *amplio* que *U*, se produce una conversión:
  - Widening conversion:
    - T y U son nombres de clases y U es superclase de T
    - T y U son nombres de interfaces y U es superinterfaz de T
    - T es una clase que implementa la interfaz U
    - (T es una subclase de una clase que implementa una interfaz que es subinterfaz de U).
- Son todas verificables por el compilador.

Reglas para asignar en forma válida (cont.)

- Ejemplo: “a” es de tipo interfaz y el tipo de “b” es subclase de una clase que implementa una subinterfaz del tipo de “a”, puedo escribir: “a=b;”
- Ejemplo:

```
interface I { void m(); }
interface J extends I { … }
class C implements J { void m() { /*Código de m */} }
class D extends C { void m() { /* Redefinición de m */} }
```

Entonces si tengo las declaraciones:

“I a;” y “D b;” es posible escribir “a = b;” y escribir “a.m()” ejecutará el código de “m” escrito en la clase “D” por resolución dinámica.

// I.java	// C.java

**public interface I {
public void m();
}**
	**public class C implements J {
	public void m() {
	System.out.println("Estoy en método m de clase C");
	}
	
	public void n() {
	System.out.println("Estoy en método n de clase C");
	}
	}**

// J.java
public interface J extends I {
public void n();
}
	// D.java
	public class D extends C {
	public void m() { // Redefino el comportamiento de m
	System.out.println("Estoy en método m de clase D");
	}
	}

```
public class Test {
    public static void main(String [] args) {
            I a;   // “a” es de tipo interfaz “I”
            D b = new D();  // “b” es de tipo clase “D”

            a = b;  // Asignación válida
            a.m();    // Imprime "Estoy en método m de clase D“
            // porque en ejecución “a” apunta a un objeto de tipo “D”
            // a.n(); // No compila porque "I" no tiene método "n()"
    }
}
```

Narrowing conversions

Ocurre cuando un tipo T se convierte en un tipo S más *angosto*.

Supongamos “S a;” y “T b;” y quiero escribir “a = b;”.

- Casos comunes:
  T y S son clases y S es subclase de T.
- T y S son interfaces y S es subinterfaz de T.
- T es una interfaz implementada por la clase S.

Se requiere casting explícito (e.g. “a = (S) b;”) y su correctitud va a ser controlada en ejecución.

La falla produce una ClassCastException.

Clases Wrapper

- En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, float, char, etc.) como objetos.
- Las clases wrapper existentes son, Byte para byte, Short para short, Integer para int, Long para long, Boolean para boolean, Float para float, Double para double y Character para char.
- Observe que las clases envoltorio tienen siempre la primera letra en mayúsculas.
- Nota: Las classes wrapper no están pensadas para actualizar los valores envueltos.
- Las clases wrapper se usan como cualquier otra:

```
Integer i = new Integer(5);
int x = i.intValue();
```

Notas

Autoboxing: Conversión automática entre tipo primitive y tipo wrapper.

```
int i = 8;
Integer x = i;  // int se convierte en Integer
```

Unboxing: Conversión automática entre tipo wrapper y tipo primitivo.

```
int j  = x;  // Integer se convierte en int.
```

Ojo: Las clases wrapper no sirven para simular pasaje de parámetros por referencia.

Genericidad (pre Java 5)

- Problema: Implementar una colección genérica.
- Para ello utilizaremos como estructura de datos subyacente un arreglo de objects

```
public class Coleccion
{
        protected int cant;    // cantidad de elementos almacenados
        protected Object [] datos;  // arreglo de elementos

        public Coleccion(int n)
        {
                cant = 0; // Hay 0 elementos al crear la coleccion
                datos = new Object[n];
        }
```

Genericidad (pre Java 5)

```
public void insertar(Object item) throws ColeccionException {
    if( cant == datos.length )
            throw new ColeccionException( "insertar: Coleccion llena" );
    datos[cant++] = item;
}

 public Object recuperar(int i) throws ColeccionException {
    if( i<0 || i>=cant )
            throw new ColeccionException( "recuperar: Elemento " + i +
                        " inexistente" );
    return datos[i];
}
} // Fin de la clase
```

```
public class Test {  // Archivo Test.java
  public static void main( String [] args ) {
   try {
    // conj_int es una colección de enteros:
     Coleccion conj_int = new Coleccion(5);
     conj_int.insertar( 1 );
     conj_int.insertar( 2 );
     int i = (Integer) conj_int.recuperar( 1 );
     System.out.println( "x: " + i );  // Imprime x: 2

     // conj_ch es una colección de caracteres:
     Coleccion conj_ch = new Coleccion(5);
     conj_ch.insertar( 'a' );
     conj_ch.insertar( 'b' );
     char c = (Character) conj_ch.recuperar( 1 );
     System.out.println( "c: " + c ); // Imprime c: b
```

```
        // ¡Conj es una colección heterogénea!
        Coleccion conj = new Coleccion(5);
        conj.insertar( 1 );
        conj.insertar( 'a' );
        conj.insertar( true );
        char d = (Character) conj.recuperar(1);
        System.out.println( "d: " + d ); // Imprime d: a

        // Esta sentencia lanza una excepción ClassCastExcepcion a
        // propósito al recuperar un entero y pensar que era un carácter:
        System.out.println( (Character) conj_int.recuperar(1));

} catch( ColeccionException e ) {
        System.out.println( "main: " + e.getMessage() );
        e.printStackTrace();
} catch( ClassCastException e ) {
        e.printStackTrace();    // Imprime java.lang.ClassCastException:
// java.lang.Integer cannot be cast to java.lang.Character
//      at Test.main(Test.java:28)
}  }  /* Fin de main */ } /* Fin de la clase */
```

Genericidad (pre Java 5)

Problemas del acercamiento:

- Hay que realizar castings explícitos para recuperar elementos de un conjunto.
- No hay manera de garantizar la homogeneidad de los elementos del conjunto
- Ej:

```
Conjunto conj = new Conjunto(5);
conj.insertar( 1 );
conj.insertar( 'a' );
conj.insertar( true );
char d = (Character) conj.recuperar(1);
System.out.println( "d: " + d );
```

Genericidad paramétrica (Java 5 y posterior) ☺

- Java 5.0 incluye un framework de genericidad para usar tipos de dato abstracto que evitan casting explícito.
- Un <u>tipo genérico</u> es un tipo que no es definido en compilación sino en tiempo de ejecución y
- permite definir una clase en función de <u>parámetros formales de tipo</u> que abstraen los tipos de algunos datos internos de una clase.
- Dada una clase genérica, vamos instanciar objetos usando <u>parámetros actuales de tipo</u>.

Generidad paramétrica

- Problema: Programar una clase para representar un par ordenado que permita almacenar dos valores de cualquier tipo.
- La clase será genérica, se llamará Pair y estará parametrizada en términos de los tipos de las dos componentes del par.

```java
/* Implementa un par ordenado (clave, valor) */
public class Pair<K,V>   // K y V son parámetros formales de tipo.
{
    // Los parámetros formales de tipo se pueden usar
    // para declarar atributos y variables:
    protected K key;    // key es tipo K
    protected V value;    // value es de tipo V

    // Los parámetros formales de tipo se pueden usar como
    // tipo en un parámetro formal:
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // Los parámetros formales de tipo se pueden usar como tipo
    // de retorno de una función
    public K getKey() { return key; }
    public V getValue() { return value; }
    public String toString() {
        return "[" +  getKey() +
        "," + getValue() + "]";
    }
}
```

Usando la clase Pair

- Utilizaremos la clase par ordenado para crear:
  - un par de tipo (string, número real) y le almacenaremos el par (“Sergio”, 3.14)
  - Un par de tipo (entero, carácter) y le almacenaremos el par (8, ‘b’)
  - Un par de tipo (persona, persona) y le almacenaremos a Juan de 20 años como primera componente y a Tito de 40 años como segunda.

```java
public class TestPair {
    public static void main( String [] args ) {
        try{
            Pair<String, Float> p = new Pair<String, Float>("Sergio", 3.14f );
            Pair<Integer, Character> q = new Pair<Integer, Character>(8, 'b');
            Pair<Persona, Persona> r = new Pair<Persona, Persona>(
                new Persona("Juan",20), new Persona("Tito", 40)
                );
            System.out.println( "p: " + p );
            System.out.println( "q: " + q );
            System.out.println( "r: " + r );
        } catch(PersonaException e ) {
            System.out.println( "main: " + e.getMessage() );
        }
    }
}
```

```
>java TestPair
p: [Sergio,3.14]
q: [8,b]
r: [Juan 20,Tito 40]

>_
```

Notas:

1) Se deben usar tipos de clase como parámetro actual de tipo (i.e., no tipos básicos)
2) Observe cómo funciona el autoboxing en los constructores.

Conjunto con genericidad paramétrica

- <u>Problema:</u> Implementar una colección genérica.
- Para ello utilizaremos como estructura de datos subyacente un arreglo de elementos de tipo E
- Luego crearemos un conjunto de enteros con el contenido { 1, 2 }.
- Luego crearemos otro conjunto de caracteres con los elementos { a, b }.

```java
// Archivo: ColeccionException.java:
public class ColeccionException extends Exception {
	public ColeccionException( String msg ) {
		super( msg );
	}
}


// Archivo: Coleccion.java (para completar con más operaciones):
// Note que la interface es genérica, tiene un parámetro formal de tipo E
// que indica el tipo de los elementos de la colección.
public interface Coleccion<E> {
	// Permite insertar item al final de la colección.
	public void insertar(E item) throws ColeccionException;

	// Permite recuperar el elemento i-esimo de la colección
	public E recuperar(int i) throws ColeccionException;
}
```

```
// Archivo fuente: ColeccionConArreglo.java
public class ColeccionConArreglo<E> implements Coleccion<E>
{
	protected int cant;    // cantidad de elementos almacenados
	protected E [] datos;  // arreglo de elementos

	public ColeccionConArreglo(int n)
	{
		cant = 0; // Hay 0 elementos al crear la colección
		datos =  (E []) new Object[n]; // produce warning
	}

	public void insertar(E item) throws ColeccionException
	{
		if( cant == datos.length )
			throw new ColeccionException(
						"insertar: Colección llena" );
		datos[cant++] = item;
	}
```

// Archivo fuente: ColeccionConArreglo.java continuación:

```
        public E recuperar(int i) throws ColeccionException
        {
                if( i<0 || i>=cant )
                        throw new ColeccionException(
                            "recuperar: Elemento" + i + " inexistente" );
                return datos[i];
        }
}
```

```
// Archivo fuente: Test.java (nueva versión)
public class Test
{
  public static void main( String [] args ) {
    try {
        // Creo una colección de enteros.
        // Debo usar la clase wrapper Integer porque los parámetros
        // actuales de tipo no pueden ser tipos básicos.
        // Note como asignamos un objeto de tipo clase
        // ColeccionConArreglo  a una variable de tipo interfaz Colección:
        Coleccion<Integer> conj_int =
            new ColeccionConArreglo<Integer>(5);

        conj_int.insertar( 1 ); // autoboxing de int a Integer
        conj_int.insertar( new Integer(2) ); // Alternativa redundante

        int i = conj_int.recuperar( 1 ); // Sin casting!
        System.out.println( "x: " + i );  // Imprime x: 2
```

```
// Archivo fuente: Test.java (nueva versión)… continuación

        // Creo una colección de caracteres.
        Coleccion<Character> conj_ch =
              new ColeccionConArreglo<Character>(5);
        conj_ch.insertar( 'a' );  // Autoboxing de char a Character
        conj_ch.insertar( 'b' );
        char c = conj_ch.recuperar( 1 );
        // Sin casting! + unboxing de Character a char (i.e. recuperar
        // retorna un Character que es convertido a char
        // para hacer la asignación).
        System.out.println( "c: " + c ); // Imprime: c : b
} catch( ColeccionException e ) {
        System.out.println( "main: " + e.getMessage() );
        e.printStackTrace();
}  // Fin de catch
} // Fin de main
} // Fin de clase Test.
```

Sumario

- Horarios
- Contexto curricular
- Objetivos
- Factores de calidad
- Excepciones
- Interfaces
- Clases Wrapper
- Genericidad

- Clase basada en Capítulos 1 y 2 de Goodrich y Tamassia