Universidad Tecnológica de Pereira

Mision TIC 2022

El futuro digital es de todos

MinTIC

Unidad 2

06-Colecciones y Excepciones
Pruebas de código

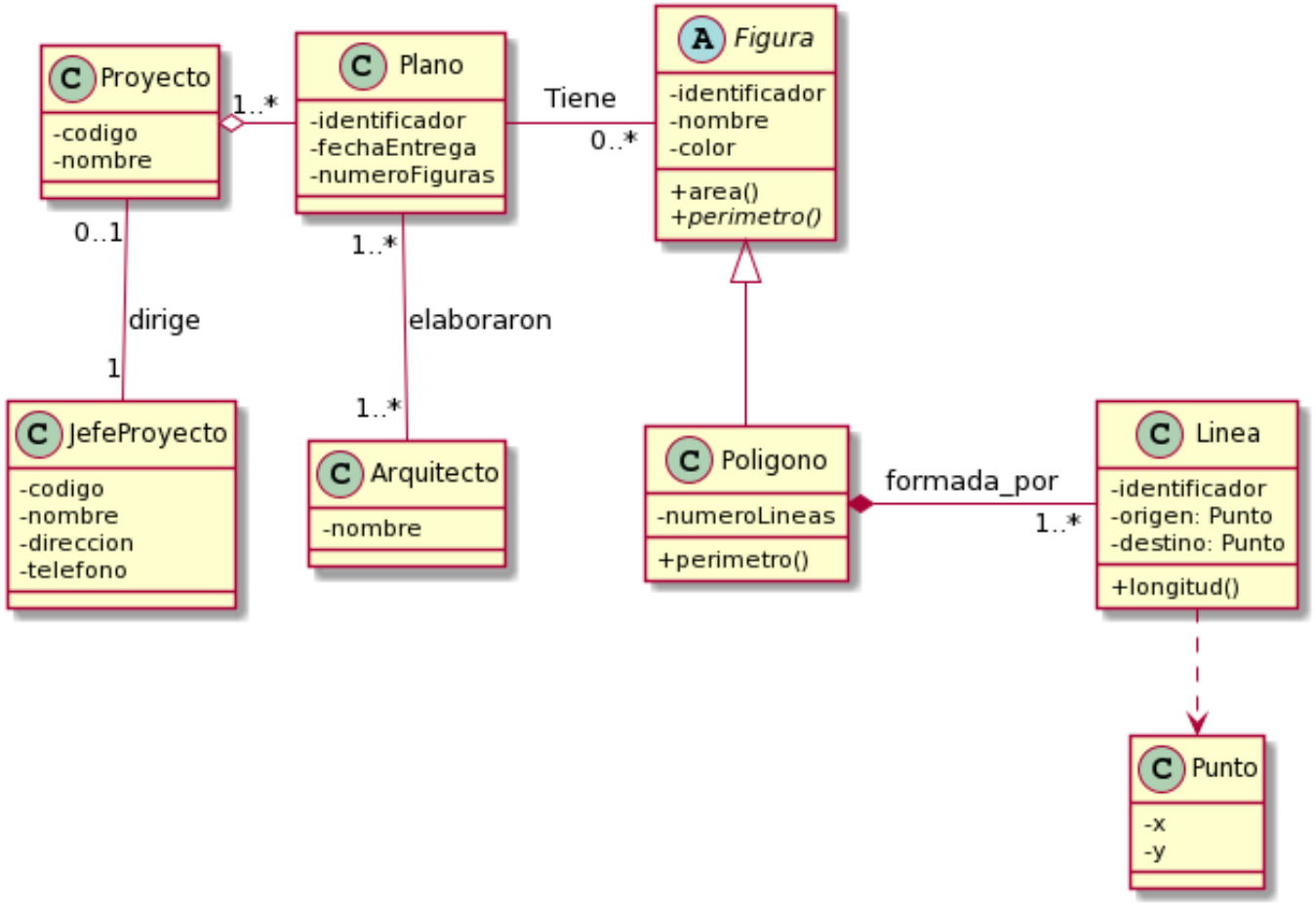
Hechos QUE CONECTAN

Retomando: Proyecto

Un estudio de arquitectura desea crear una base de datos para gestionar sus proyectos. Nos dan las siguientes especificaciones:

- Cada proyecto tiene un código y un nombre. Un proyecto tiene uno y solo un jefe de proyecto y un jefe de proyecto sólo puede estar involucrado en un proyecto o en ninguno.
- De cada jefe de proyecto se desean recoger sus datos personales (código, nombre, dirección y teléfono). Un jefe de proyecto se identifica por un código. No hay dos nombres de jefe de proyecto con el mismo nombre.
- Un proyecto se compone de una serie de planos, pero éstos se quieren guardar de modo independiente al proyecto. Es decir, si en un momento dado se dejara de trabajar en un proyecto, se desea mantener la información de los planos asociados.
- De los planos se desea guardar su número de identificación, la fecha de entrega, los arquitectos que trabajan en él y un dibujo del plano general con información acerca del número de figuras que contiene.
- Los planos tienen figuras. De cada figura se desea conocer, el identificador, el nombre, el color, el área y el perímetro. Además, de los polígonos se desea conocer el número de líneas que tienen, además de las líneas que lo forman. El perímetro se desea que sea un método diferido; el área se desea implementarlo como genérico para cualquier tipo de figura, pero además se desea un método específico para el cálculo del perímetro de los polígonos.
- De cada líneas que forma parte de un polígono se desea conocer el punto de origen y el de fin (según sus coordenadas, X e Y), así como la longitud. Cada línea tiene un identificador que permite diferenciarlo del resto. La longitud de la línea se puede calcular a partir de sus puntos origen y final.

Solución Proyecto



Universidad Tecnológica de Pereira

Mision TIC 2022

El futuro digital es de todos

MinTIC

Java Collection Framework

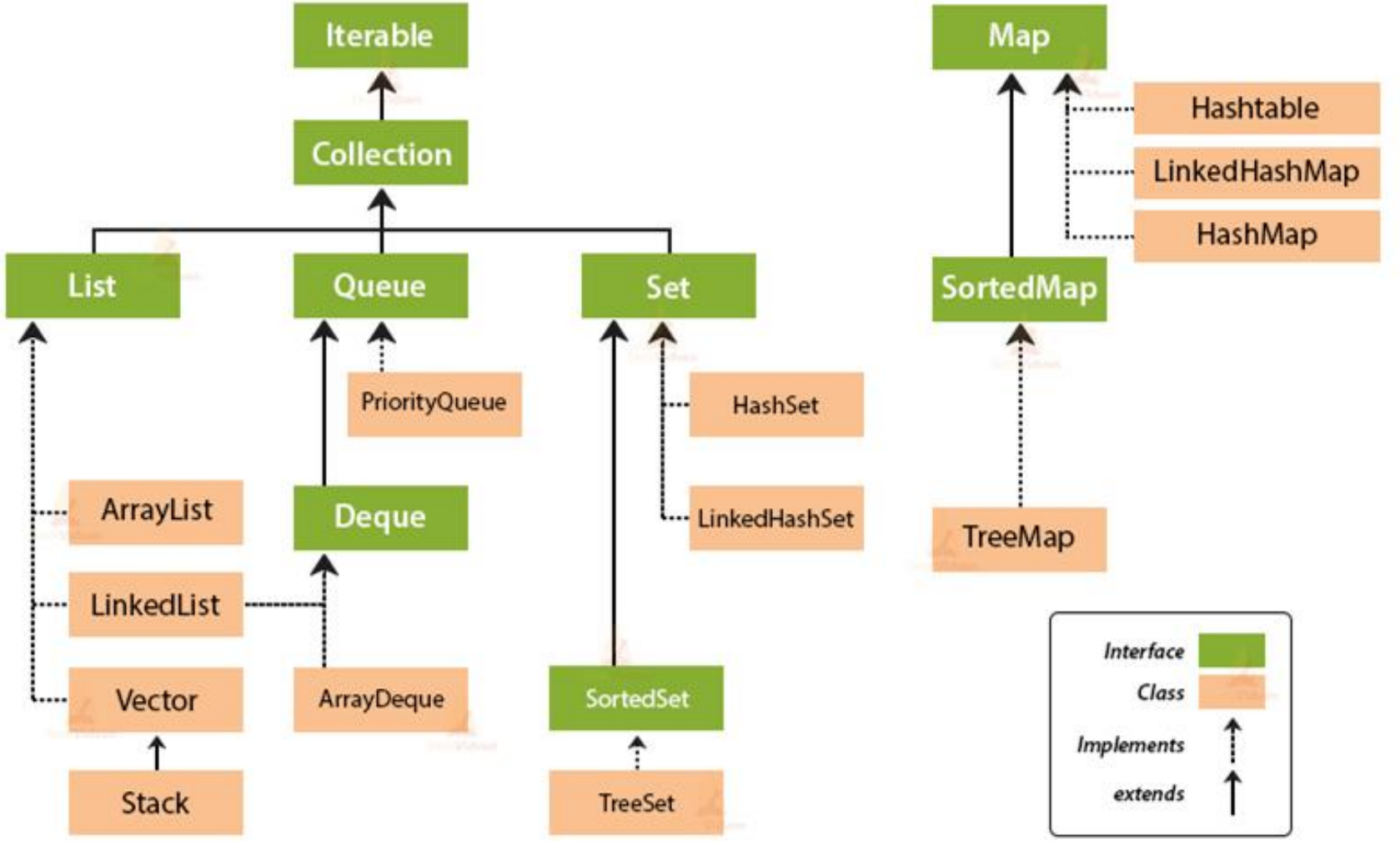
https://docs.oracle.com/javase/tutorial/collections/index.html

Hechos QUE CONECTAN

Java Collection Framework

- Es como se conoce a la librería de clases contenedoras de Java que podemos encontrar en el paquete estándar `java.util`.
- Estas clases sirven para almacenar colecciones de objetos, como listas, conjuntos, mapas, ...
- Todas estas clases permiten guardar en ellas referencias a objetos

```
// Lista de enteros.
// Puede haber enteros repetidos en la lista:
List<Integer> listaDeManzanas;

// Conjunto de enteros.
// No puede haber enteros repetidos:
Set<Integer> conjuntoDeNaranjas;

// Un mapa que asocia a una cadena un entero,
// como en una lista de notas de un examen:
//    [("Juan Goytisolo", 9.5),
//     ("Pablo Iglesias", 5.0), ...]
Map<String, Integer> mapaDeNotas;
```


Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet
Map
Hashtable
LinkedHashMap
HashMap
SortedMap
TreeMap
Interface
Class
Implements
extends

Listas

- Llamamos lista a cualquier colección de objetos ordenados por posición, como en un array.
- En una lista podemos añadir elementos, acceder a ellos por su posición en la lista, eliminar elementos de la lista y otras operaciones, como vaciar la lista, copiarla, etc.
- En una lista puede haber objetos repetidos, es decir, objetos que son iguales según el método equals() de su clase.

Implementaciones:

- **ArrayList**
  Guarda una lista de elementos en un array de tamaño dinámico.
- **LinkedList**
  Permite inserción y borrado de elementos de la lista.
- **Vector**
  Aumenta o reduce su tamaño de forma dinámica.
- **Stack**
  Representa una pila de objetos de último en entrar, primero en salir (LIFO).

Operaciones de Listas

- **Crear una lista**

```
List<Integer> listaEnteros = new ArrayList<>();
var listaEnteros = new ArrayList<Integer>();
```

- **Añadir elementos a la lista**

```
listaEnteros.add(4);
listaEnteros.add(5);
listaEnteros.add(7);
listaEnteros.add(2, 6); // Agrega 6 entre 5 y 7
```

- **Tamaño de la lista**

```
listaEnteros.size(); // 4
```

- **Está en la lista?**

```
listaEnteros.contains(7); // 'true'
listaEnteros.contains(8); // 'false'
```

- **Posición de un objeto**

```
listaEnteros.indexOf(6); // 2
listaEnteros.indexOf(10); // -1
```

- **Obtener elementos de la lista**

```
var primero = listaEnteros.get(0); // 4
var ultimo =
    listaEnteros.get(listaEnteros.size() - 1); // 7
```

- **Eliminar un objeto**

```
listaEnteros.remove(new Integer(7)); // true
listaEnteros.remove(1); // 5
```

- **Está vacía?**

```
listaEnteros.isEmpty(); // false
listaEnteros.clear(); // Ahora si está vacía
```

Recorrer una lista

- **Bucle convencional**

```
for (int i=0; i < listaEnteros.size(); i++) {
  System.out.println(listaEnteros.get(i));
}
```

- **Foreach**

```
for (Integer entero: listaEnteros) {
  System.out.println(entero);
}
```

- **Iterador**

```
var iterador = listaEnteros.iterator();
while (iterador.hasNext()) {
  Integer entero = iterador.next();
  System.out.println(entero);
}
```

- **forEach**

```
listaEnteros.forEach((entero) -> {
  System.out.println(entero);
});
listaEnteros.forEach(System.out::println);
```

- **Stream Foreach**

```
listaEnteros.stream().forEach((entero) -> {
  System.out.println(entero);
});

listaEnteros.stream()
            .forEach(System.out::println);
```

Conjuntos

- Llamamos conjunto a cualquier colección de objetos de la misma clase sin ningún orden en particular.
- Cada elemento sólo aparece una vez, al contrario que en una lista, donde podían repetirse.

Implementaciones:

- **HashSet**
  Usa tablas hash para acelerar la búsqueda, adicion y eliminacion de elementos.
- **LinkedHashSet**
  Es una versión ordenada de HashSet que mantiene una lista doblemente vinculada en todos los elementos..
- **TreeSet**
  Usa un árbol binario para acelerar la búsqueda, adicion y eliminacion de elementos.

Operaciones de Conjuntos

- **Crear un conjunto**

```
Set<Integer> conjuntoEnteros = new HashSet<>();
var conjuntoEnteros = new HashSet<Integer>();
```

- **Añadir elementos al conjunto**

```
conjuntoEnteros.add(4);
conjuntoEnteros.add(5);
conjuntoEnteros.add(7);
conjuntoEnteros.add(4); // retorna false por repetido
```

- **Tamaño del conjunto**

```
conjuntoEnteros.size(); // 3
```

- **Está en el conjunto?**

```
conjuntoEnteros.contains(7); // 'true'
conjuntoEnteros.contains(8); // 'false'
```

- **Eliminar un objeto**

```
conjuntoEnteros.remove(new Integer(7)); // true
```

- **Está vacía?**

```
conjuntoEnteros.isEmpty(); // false
conjuntoEnteros.clear(); // Ahora si está vacía
```

- **Recorrer todos los elementos**

```
for (Integer entero: conjuntoEnteros) {
  System.out.println(entero);
}
```

Colas

- La Cola (Queue) se utiliza para insertar elementos al final de la cola y los elimina desde el principio de la cola.
- El Deque representa una cola de dos extremos, es decir, una cola en la que puede agregar y eliminar elementos de ambos extremos.

Implementaciones:

- **PriorityQueue**
  Implementación del algoritmo de cola.
- **LinkedList**
  Permite inserción y borrado de elementos de la lista. Implementa la interface Deque.
- **ArrayDeque**
  Proporciona una forma de aplicar array de tamaño dinámico, además de la implementación de la interfaz Deque.

Operaciones de Cola

- **Crear una Cola**

```
Queue<Integer> colaEnteros = new PriorityQueue<>();
var colaEnteros = new LinkedList<Integer>();
```

- **Añadir elementos al conjunto**

```
colaEnteros.add(4);
colaEnteros.offer(5);
colaEnteros.offer(7);
colaEnteros.offer(6);
```

- **Tamaño de la Cola**

```
colaEnteros.size(); // 4
```

- **Obtener un elemento de la Cola**

```
colaEnteros.poll(); // 4
colaEnteros.remove(); // 5
```

- **Obtener el primer elemento de la Cola**

```
colaEnteros.peek(); // 7
colaEnteros.element(); // 7
```

- **Eliminar un objeto**

```
colaEnteros.remove(); // 7
```

- **Está vacía?**

```
colaEnteros.isEmpty(); // false
colaEnteros.clear(); // Ahora si está vacía
```

- **Recorrer todos los elementos**

```
for (Integer entero: colaEnteros) {
  System.out.println(entero);
}
```

Mapas

- Los mapas permiten establecer una correspondencia entre pares de objetos: uno que actúa como clave y otro como valor asociado a esa clave.
- Un diccionario es un mapa entre cadenas de texto: la palabra que buscamos en el diccionario actúa como clave y su significado como valor asociado.

Implementaciones:

- **Hashtable**
  Es una estructura de datos que utiliza una función *hash*. Ya no es usado..
- **HashMap**
  Almacena pares clave / valor en una tabla hash, y no están ordenados de ninguna manera..
- **LinkedHashMap**
  Agrega una lista vinculada a la estructura del HashMap.
- **TreeMap**
  Es implementado como un árbol Rojo-Negro, un tipo de árbol binario de búsqueda equilibrada..

Operaciones de Mapas

- **Crear un mapa**

```
Map<CoordenadaAjedrez, PiezaAjedrez> damero = new HashMap<>();
var damero = new HashMap<CoordenadaAjedrez, PiezaAjedrez>();
```

Añadir elementos al mapa

```
damero.put(new CoordenadaAjedrez('A',3),
           new PiezaAjedrez("ALFIL",Color.BLANCO));

damero.put(new CoordenadaAjedrez('B',7),
           new PiezaAjedrez("CABALLO",Color.NEGRO));

damero.put(new CoordenadaAjedrez('B',7),
           new PiezaAjedrez("REINA",Color.BLANCO));
```

- **Tamaño del mapa**

```
damero.size(); // 3
```

- **Está en la lista?**

```
damero.containsKey(new CoordenadaAjedrez('A',3));
damero.containsValue(
           new PiezaAjedrez("ALFIL",Color.BLANCO));
```

- **Eliminar un objeto**

```
damero.remove(new CoordenadaAjedrez('A',3));
```

- **Está vacía?**

```
damero.isEmpty(); // false
damero.clear(); // Ahora si está vacía
```

- **Recorrer todos los elementos**

```
var coordenadas = damero.keySet();

for (CoordenadaAjedrez coord : coordenadas) {
  PiezaAjedrez pieza = damero.get(coord);
  System.out.println(coord + " -> " + pieza);
}
```

Universidad Tecnológica de Pereira

Mision TIC 2022

El futuro digital es de todos

MinTIC

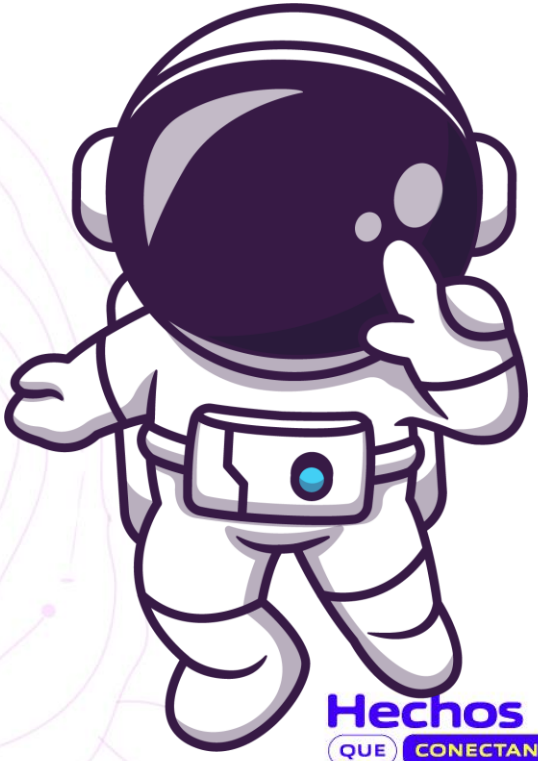
Java Generics

https://docs.oracle.com/javase/tutorial/extra/generics/index.h

Hechos QUE CONECTAN

¿Qué es Generics?

- Los generics fueron introducidos en la versión 5 de Java en 2004.
- Las colecciones (Listas, Conjuntos, etc) pueden guardar *Objects* de cualquier tipo.
- Restringe los *Objects* a ser puestos en la colección.
- Permiten al compilador informar de muchos errores de compilación que hasta el momento solo se descubrirían en tiempo de ejecución



Cómo funciona?

```
List lista = new ArrayList();

lista.add("Test");

Integer entero = (Integer) lista.get(0); // Conversión en en ejecución
```

Este error solo se mostrará cuando estemos ejecutando el programa en tiempo de ejecución.

```
List<String> lista = new ArrayList<>();

lista.add("Test");

Integer entero = (Integer) lista.get(0); // Error en compilación
```

Ejemplos

```
public class Box<T> {

  private T t;

  public T get() {
    return t;
  }

  public void set(T t) {
    this.t = t;
  }

}



var integerBox = new Box<Integer>();

var doubleBox = new Box<Double>();
```

```
public interface Pair<K, V> {
  public K getKey();
  public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

  private K key;
  private V value;

  public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
  }

  public K getKey() { return key; }

  public V getValue() { return value; }

}



var evenPair = new OrderedPair<String, Integer>("Even", 8);
```

Convenciones

Según las convenciones los nombres de los parámetros de tipo usados comúnmente son los siguientes:

- E: elemento de una colección.
- K: clave.
- N: número.
- T: tipo.
- V: valor.
- S, U, W etc: para segundos, terceros y cuartos tipos.

Métodos genéricos

```
public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {

  return p1.getKey().equals(p2.getKey())
            && p1.getValue().equals(p2.getValue());

}
```

Para el uso del método genérico:

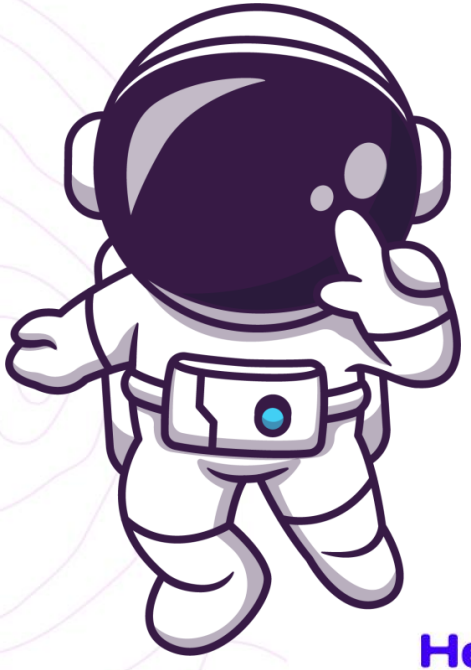
```
Pair<Integer, String> p1 = new OrderedPair<>(1, "apple");
Pair<Integer, String> p2 = new OrderedPair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

Dejando que el compilador infiera los tipos

```
boolean same = Util.compare(p1, p2);
```

Genéricos de un tipo

```
public class BoxBounds<T extends Number> {

  private T t;

  public void set(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public <U extends String> void inspect(U u){
    System.out.println("T: " + t.getClass().getName());
    System.out.println("U: " + u.getClass().getName());
  }

  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    integerBox.set(new Integer(10));
    integerBox.inspect("some text");
  }

}
```



Restricciones

- No se pueden instanciar tipos genéricos con tipos primitivos.
- No se pueden crear instancias de los parámetros de tipo.
- No se pueden declarar campos static cuyos tipos son parámetros de tipo.
- No se pueden usar casts o instanceof con tipos parametrizados.
- No se pueden crear arrays de tipos parametrizados.
- No se pueden crear, capturar o lanzar tipos parametrizados que extiendan de Throwable.
- No se puede sobrecargar un método que tengan la misma firma que otro después del type erasure.

Universidad Tecnológica de Pereira

Mision TIC 2022

El futuro digital es de todos

MinTIC

Excepciones

https://docs.oracle.com/javase/tutorial/essential/exceptions/index.ht

Hechos QUE CONECTAN

Excepciones, o sencillamente problemas.

- Es un evento que ocurre durante la ejecución de un programa, y que interrumpe el flujo normal de instrucciones.
- Sirven para informar que se ha producido una situación extraña y que debe tratarse.
- Mejor que comprobar valores de retorno

```
try {

  //Código que puede provocar errores

} catch(Tipo1 var1) {
  //Gestión del error de tipo Tipo1

} catch(Tipo2 | Tipo3 | Tipo4 var) {

  //Gestión del error de tipo Tipo2,
  // Tipo3 o Tipo4

} catch(TipoN  varN) {

  //Gestión del error de tipo TipoN

} finally {

  //Código de finally

}
```

Captura de Excepciones

- Es un mecanismo consistente en el uso de bloques try/catch/finally.
- El manejo de excepciones se logra con el bloque try.
- El bloque try puede manejar múltiples excepciones.
- La cláusula finally es ejecutada con posterioridad cualquiera sea la condición de término del try (sin o con error). Esta sección permite dejar las cosas consistentes antes del término del bloque try.

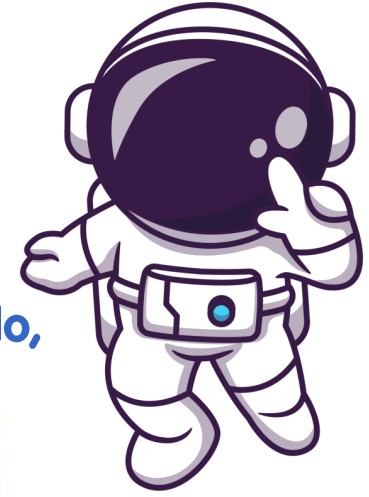
```
try {

  //Código que puede provocar errores

} catch(Tipo1 var1) {
  //Gestión del error de tipo Tipo1

} catch(Tipo2 | Tipo3 | Tipo4 var) {

  //Gestión del error de tipo Tipo2,
  // Tipo3 o Tipo4

} catch(TipoN  varN) {

  //Gestión del error de tipo TipoN

} finally {

  //Código de finally

}
```

Excepciones, o sencillamente problemas.

```
public static void main(String arg[]) {

  int [] array = new int[20];
  try {
    array[-3] = 24;
    int b = 0;
    int a = 23/b;
  } catch(ArrayIndexOutOfBoundsException | ArithmeticException ex) {
    System.out.println("Error de índice en un array");
  }

}
```

Ventajas

- Separación entre el "código normal" y el código de manejo de errores
- Propagación de errores hacia "arriba" en el stack de llamadas (y detención automática del programa si una situación de error no es manejada)
- Facilidades para la agrupación de tipos de errores
- Facilidades para entregar información del error producido, sin que se produzca interferencia con el retorno normal



Características

1. Limpieza de código
2. Propagación de errores
3. Agrupación de errores

```
leerArchivo() {

  try {

    abrir el archivo;
    determinar su tamaño;
    crear la memoria necesaria;
    leer el archivo a memoria;
  } catch (error al abrir archivo) {
    ...
  } catch (error al obtener tamaño archivo) {
    ...
  } catch (error al crear memoria) {
    ...
  } catch (error al leer archivo) {
    ...
  } catch (error al cerrar archivo) {
    ...
  } finally {
    cerrar el archivo;
  }
}
```

Características

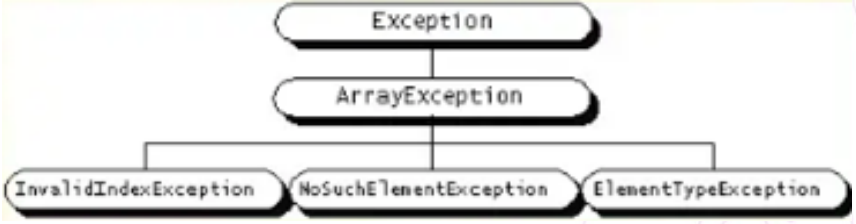
1. Limpieza de código
2. Propagación de errores
3. Agrupación de errores

Si un método no atrapa(catch) una excepción, el método aborta, propagando la excepción.

```
void metodo1() {
  try {
    metodo2();
    ...
  } catch (Exception ex) {
    manejar el error;
  }

}

void metodo2() {
  metodo3();
  …
}

void metodo3() {
  leerArchivo();
   ...

}
```

Características

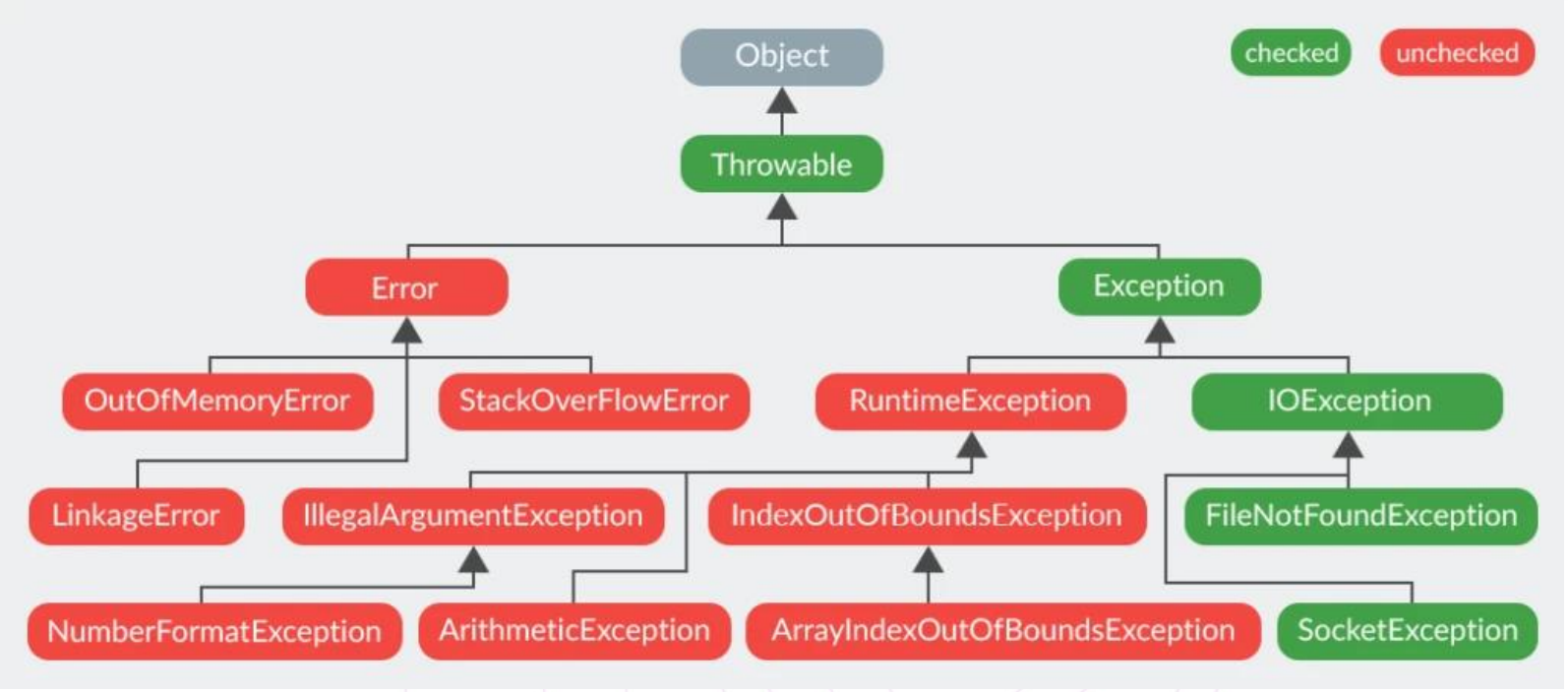
1. Limpieza de código
2. Propagación de errores
3. Agrupación de errores




```
void metodo4() {
  try {
    ...
  } catch (InvalidIndexException ex) {
    manejar el error;
  }

}

void metodo4() {
  try {
    ...
  } catch (Exception ex) {
    manejar el error;
  }

}
```

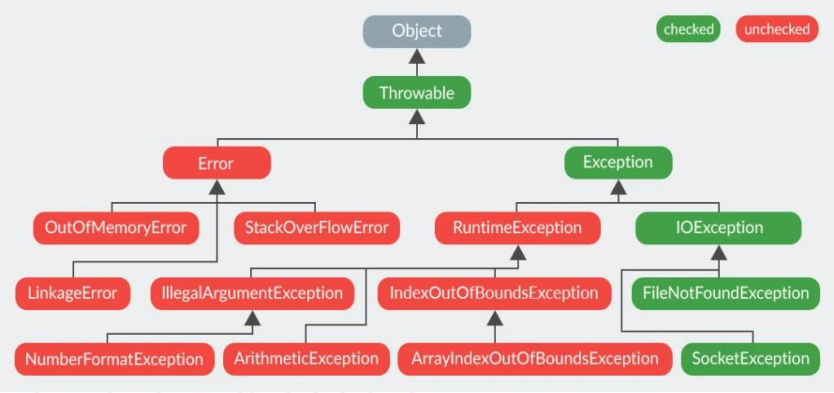
Tipos de Excepciones

- Las hay de dos tipos
  - Aquellas generadas por el lenguaje Java. Éstas se generan cuando hay errores de ejecución, como al tratar de acceder a métodos de una referencia no asignada a un objeto, división por cero, etc.
  - Aquellas no generadas por el lenguaje, sino incluidas por el programador.
- El compilador chequea por la captura de las excepciones lanzadas por los objetos usados en el código.
- Si una excepción no es capturada debe ser relanzada.


Object
checked
unchecked
Throwable
Error
Exception
OutOfMemoryError
StackOverFlowError
RuntimeException
IOException
LinkageError
IllegalArgumentException
IndexOutOfBoundsException
FileNotFoundException
NumberFormatException
ArithmeticException
ArrayIndexOutOfBoundsException
SocketException

Checked y Uncheked

- **Excepciones "checked"**
  - Si un método genera (throw) o propaga una excepción checked, debe declararlo (throws) en su firma.
- **Excepciones "unchecked"**
  - No es necesario que un método declare (throws) las excepciones unchecked que genera (throw) o propaga (aunque puede hacerlo)



Lanzando Excepciones

- Siempre es posible lanzar alguna excepción de las ya definidas en Java .
- Para que un *método* pueda propagar una excepción es necesario decirle los tipos de excepción que puede lanzar luego de la palabra throws.
- Para lanzar una excepción, se usa la palabra throw y el objeto de excepción a lanzar.

```
public void doio1 (InputStream in, OutputStream out)
    throws IOException {
  int c;
  while ((c=in.read()) >=0 ) {
    c = Character.toLowerCase((char) c);
    out.write( c );
  }
}

public void doio2 (InputStream in, OutputStream out)
    throws Throwable {
  int c;
  try {
    while ((c=in.read()) >=0 ) {
      c = Character.toLowerCase((char) c);
      out.write( c );
    }
  } catch (Throwable t) {
    throw t;
  }
}
```

Creando nuevas Excepciones

- También se puede definir nuevas excepciones creando clases derivadas de las clases Error o Exception.
- Para definir excepciones checked, lo aconsejable es derivarlas de la clase *Exception*
- Para definir excepciones unchecked, lo aconsejable es derivarlas de la clase *RuntimeException*

```
class ZeroDenominatorException extends Exception {
  private int n;
  public ZeroDenominadorException () {}
  public ZeroDenominadorException(String s) {
    super(s);
  }
  public setNumerator(int _n) { n = _n}
}
...
public Fraction (int n, int d)
                    throws ZeroDenominatorException {
  if (d == 0) {
    var myExc = new ZeroDenominatorException(
          "Fraction: Fraction with 0 denominator?");

    myExc.setNumerator(n);
    throw myExc;

  }
  ...
}
```

Universidad Tecnológica de Pereira

Mision TIC 2022

El futuro digital es de todos

MinTIC

Pruebas de Software

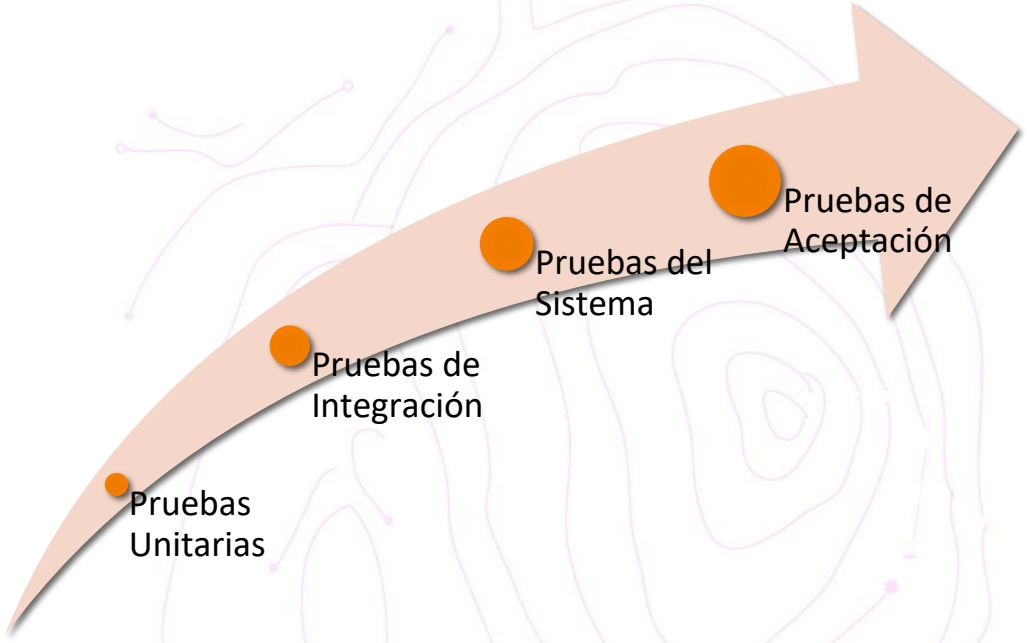
Hechos QUE CONECTAN

¿Qué es un caso de prueba?

- Conjunto de condiciones y variables bajo las cuales el analista (o tester) va a determinar si el requisito de la aplicación es parcial o completamente satisfactorio.
- Existen dos tipos de casos:
  - **Positivos**: Muestran una funcionalidad
  - **Negativos**: Comprueban situaciones en las que hay situaciones de errores.
- Cada requisito debe tener al menos un caso de prueba.
- De cada caso de prueba se debe tener **una entrada conocida** y **una salida esperado**.
- Los casos de prueba deben ser trazables.

- **Requisito: El precio del producto debe ser mayor o igual a cero**
- **Casos de Prueba Positivos**
  - **Precio es igual a 0**
  - **Precio es igual a 10**
  - **Precio es igual a 15**
- **Caso de Prueba Negativos**
  - **Precio es igual a -5**

Niveles de Prueba



Pruebas unitarias

Prueba los componentes después de su codificación

Se corren todas las pruebas después de cada cambio para verificar que el cambio no afecta otros elementos ya entregados.

Se realiza por los desarrolladores

- ¿Se cumplen las especificaciones?
- ¿Realiza tratamiento (valida) los datos de entrada inválidos?
- ¿Se validan los valores límite?

Método de Caja Blanca

Framework más utilizado es JUnit (https://junit.org/junit5/)

Pruebas de integración

Comprueba la interacción entre los componentes que han sido integrados y detectar defectos en las interfaces.

Cada componente ha pasado las pruebas unitarias

Realizado por desarrolladores (o tester)

Busca resolver los siguientes problemas

- Pérdida de datos
- Manipulación errónea de datos
- Los componentes interpretan los datos de entrada de manera diferente

Se utilizan métodos de Caja Blanca o Negra

Pruebas de sistema

Comportamiento de todo el sistema o producto.

Se prueban requisitos funcionales y no funcionales

Se prueba el sistema desde el punto de vista del usuario.

Se debe realizar en un entorno muy similar al entorno de producción.

Se realiza por Testers (o desarrolladores)

Se utilizan métodos de Caja Negra.

Pruebas de aceptación

Verificación formal la conformidad del sistema con los requisitos.

El sistema cumple con el funcionamiento esperado y cumple con los requisitos contractuales.

Tipos

- Prueba Alfa – Aceptación interna: Se prueba por miembros de la organización donde se desarrolló el software y se hagan con los recursos del proveedor.
- Prueba Beta – Aceptación externa: Se prueba por miembros del cliente o usuarios finales.

Pirámide ideal de Pruebas de Software

Instalación de JUnit 5 (pom.xml)

```
<project … >
  . . .
  <dependencies>
    . . .
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.7.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-runner</artifactId>
      <version>5.7.2</version>
      <scope>test</scope>
    </dependency>
    . . .
  </dependencies>
  . . .
```

```
  <build>
    . . .
    <plugins>
      . . .
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.2</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.22.2</version>
      </plugin>
      . . .
    </plugins>
    . . .
  </build>
  . . .
</project>
```

https://junit.org/junit5/docs/current/user-guide/#running-tests-build-maven