Revisión del Lenguaje de Programación Java

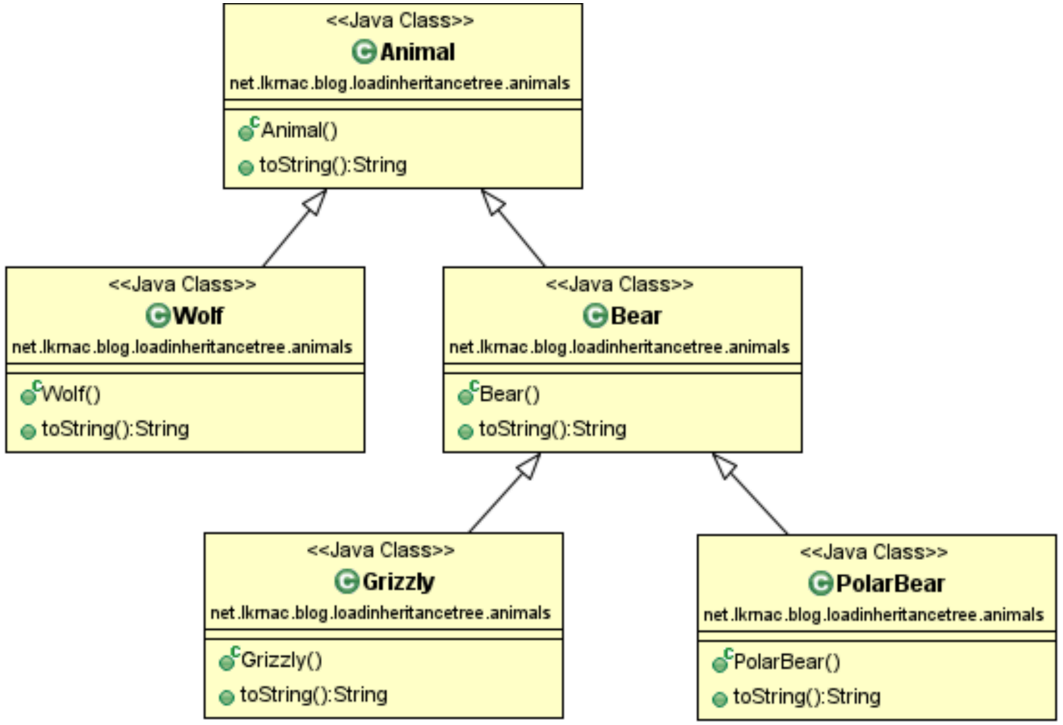
Profesores Estructuras de Datos, 2024.

Dpto. Lenguajes y Ciencias de la Computación.

University of Málaga

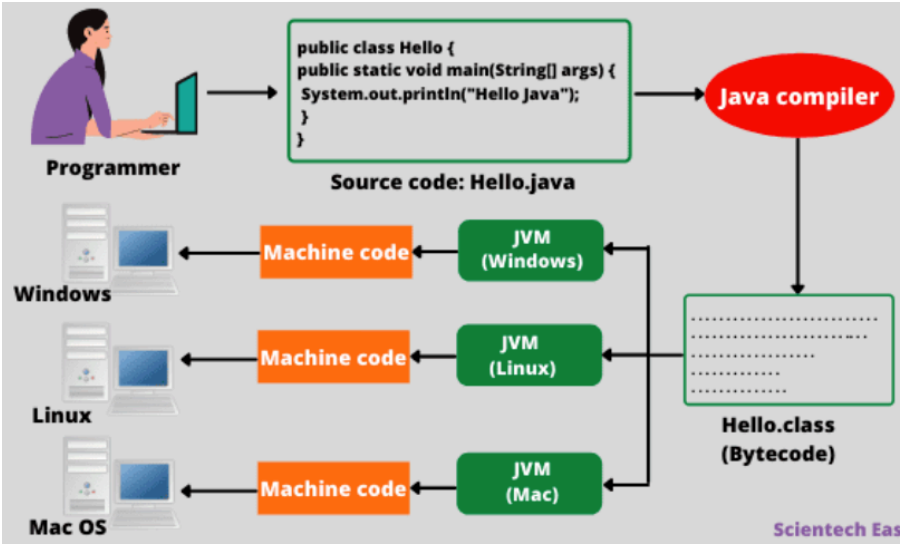
El Lenguaje de Programación Java (1/3)

- Java es un lenguaje de programación de propósito general y de alto nivel.
- Es un lenguaje basado en clases y *orientado a objetos*.



El Lenguaje de Programación Java (2/3)

- Las aplicaciones Java se compilan típicamente a *bytecode*.
- Está diseñado para permitir a los desarrolladores de aplicaciones escribir una vez, ejecutar en cualquier lugar (write once, run anywhere *WORA*).



El Lenguaje de Programación Java (3/3)

- Java es uno de los lenguajes de programación más populares en uso para el desarrollo de *aplicaciones*.

IEEE: The Top Programming Languages 2024

Clases y Objetos

- Una *clase* es un plano o plantilla para crear *objetos*.
  - Define los datos (*atributos*) y comportamientos (*métodos*) que los objetos de esa clase tendrán.
  - Los *constructores* nos permiten crear *instancias* (diferentes objetos) con datos específicos.
- Usos comunes:
  - Las clases encapsulan datos y métodos, asegurando la *integridad* de los datos y *ocultando* los detalles de implementación.
  - Las clases pueden *heredar* de otras clases, promoviendo la reutilización del código.

Ejemplo de clases y objetos

```java
public class Person {
  private String name;  // atributos de instancia
  private int age;

  public Person(String name, int age) {  // constructor de la clase
    this.name = name;
    this.age = age;
  }

  public void greet() {  // método de instancia
    System.out.println("¡Hola, soy " + name + "!");
  }
}

public class Main {
  public static void main(String[] args) {
    Person alice = new Person("Alice", 30);  // una instancia
    alice.greet();

    Person bob = new Person("Bob", 25);      // otra instancia
    bob.greet();
  }
}
```

Modificadores de acceso

- Los modificadores de *acceso* controlan la visibilidad de los miembros de la clase (atributos, métodos, constructores).
- Usos comunes:
  - *Encapsulación*: los campos `private` garantizan la privacidad de los datos.
  - *API Design*: los métodos `public` definen el contrato para el código externo.
  - *Herencia*: los miembros `protected` son accesibles por subclases.
  - *Paquete*: el acceso `default` solo es visible dentro del mismo paquete.

Atributos y métodos estáticos

- Los miembros *static* (atributos y métodos) pertenecen a la clase en sí, no a instancias individuales.
- Se *comparten* entre todos los objetos de la clase.
- Usos comunes:
  - *Métodos de utilidad*: los métodos `static` realizan tareas comunes sin referirse a los datos de una instancia.
  - *Factory Methods*: métodos `static` utilizados para crear nuevas instancias a partir de la clase.
  - *Constantes*: declarados como atributos `static final`.

Ejemplo de atributos y métodos estáticos.

```
public class MathUtils {
  // a constant
  public static final double PI = 3.14159265358979323846;

  public static int twice(int x) {  // static method
    return x + x;
  }

  public static double twice(double x) {  // static method
    return x + x;
  }
}

public class Main {
  public static void main(String[] args) {
    double n = MathUtils.twice(MathUtils.PI);
  }
}
```

Registros

- *Los registros* son clases de datos introducidas en Java 16.
- Generan automáticamente constructores, getters, métodos `equals`, `hashCode` y `toString`.
- Usos comunes:
  - *Datos inmutables*: Los registros representan datos que, una vez construidos, no se pueden modificar.
  - *Reducir código*: Menos código para clases de datos simples.

Ejemplo con registros (1/2)

```java
// x and y are immutable instance attributes.
// Automatically generated constructor, getters, equals, hashCode, and toString
public record Point(int x, int y) { }

public class Main {
  public static void main(String[] args) {
    Point p = new Point(1, 2);  // Constructor
    System.out.println(p.x());  // getter for x attribute
    System.out.println(p.y());  // getter for y attribute
    System.out.println(p);      // Point[x=1, y=2], toString method
  }
}
```

Comparación de objetos: `equals`

- El método `equals` se utiliza para comparar objetos en igualdad de *valor*.
- Las clases suelen *redefinirlo* (`override`) para comparar objetos.
- Si defines una nueva clase, debes hacer *override* para definir cómo se deben considerar iguales las instancias de tu clase.
- `hashCode` y `equals` deben ser consistentes.

Comparación de objetos: `==`

- Cuando se aplica a *tipos primitivos* ('int', 'double', etc.), compara sus *valores*.
- Cuando se aplica a objetos, compara sus *referencias* (que apunten a la **misma ubicación de memoria**).

Ejemplo de comparación de objetos `equals` e `==` (1/2)

```java
public record Point(int x, int y);

public class Main {
  public static void main(String[] args) {
   Point p1 = new Point(1, 2);
   Point p2 = new Point(1, 2);
   Point p3 = new Point(2, 3);
   System.out.println(p1.equals(p1)); // caso 1
   System.out.println(p1.equals(p2)); // caso 2
   System.out.println(p1.equals(p3)); // caso 3
   System.out.println(p1 == p2);      // caso 4
 }
}
```

¿Caso X verdadero o falso?

Comparación de objetos/primitivos `equals` e `==` (2/2)

```
public class Main {
  public static void main(String[] args) {
    Integer i1 = 3;
    Integer i2 = 3;
    Integer i3 = 190;
    Integer i4 = 190;
    int e1 = 3;
    int e2 = 3;
    int e3 = 4;
    System.out.println(i1.equals(i1)); // caso 1
    System.out.println(i1.equals(i2)); // caso 2
    System.out.println(i1.equals(i3)); // caso 3
    System.out.println(i1 == i2);      // caso 4
    System.out.println(i3 == i4);      // caso 5
    System.out.println(e1 == e2);      // caso 6
    System.out.println(i1.equals(e1));      // caso 7
    System.out.println(e1.equals(i1));      // caso 8
  }
}
```

¿Caso X verdadero o falso?

Interfaces

- Un *interfaz* (`interface`) define un contrato que las clases pueden implementar.
- Especifican la *signatura* de los métodos sin implementarlos.
- También pueden proporcionar implementaciones `default` de métodos.
- Usos comunes:
  - *Herencia múltiple*: Una clase puede implementar múltiples interfaces.
  - *Polimorfismo*: Las interfaces permiten tratar diferentes clases de manera uniforme.

Ejemplo de Interfaces.

```
public class Dog implements Animal {
  public void makeSound() { // implementation of makeSound for Dog
    System.out.println("Woof!");
  }
}

public class Cat implements Animal {
  public void makeSound() { // implementation of makeSound for Cat
    System.out.println("Meow!");
  }
}


public interface Animal {
  void makeSound();  // signature for unimplemented method
  default void breathe() {  // default method
    System.out.println("Breathing...");
  }
}
public class Main {
  static void makeSoundTwice(Animal animal) { // polymorphic method
    animal.makeSound();
    animal.makeSound();
  }
  public static void main(String[] args) {
    Animal dog = new Dog();
    dog.breathe();
    makeSoundTwice(dog);

    Animal cat = new Cat();
    cat.breathe();
    makeSoundTwice(cat);
  }
}
```

Genéricos

- *Los genéricos* permiten escribir código que funciona con diferentes tipos.
- Proporcionan *seguridad de tipo* y *reutilización*.
- Usos comunes:
  - *Colecciones*: las estructuras de datos genéricas (por ejemplo, `List<T>`) pueden almacenar elementos de cualquier tipo.

Ejemplo con genéricos.

```java
public class Box<T> {   // T stands for the type of element in box
  private T content;

  public Box(T content) {  // Constructor
    this.content = content;
  }

  public static <T> Box<T> of(T content) {  // Factory method
    return new Box<>(content);
  }

  public T getContent() {
    return content;
  }
}

public class Main {
  public static void main(String[] args) {
    Box<String> box1 = new Box<>("hello");   // here T is a String. Notice <>
    System.out.println(box1.getContent());

    Box<Integer> box2 = Box.of(123);         // here T is an Integer
    System.out.println(box2.getContent());
  }
}
```

Ejemplo de genéricos con registros (2/2)

```java
// A and B are generics. _1 and _2 are immutable instance attributes
public record Tuple2<A, B>(A _1, B _2) {
  // Factory method: Creates a new Tuple2 object
  public static <A, B> Tuple2<A, B> of(A x, B y) {
    return new Tuple2<>(x, y);
  }

  // Method: Returns a new Tuple2 with components swapped
  public Tuple2<B, A> swap() {
    return Tuple2.of(_2, _1);
  }

  public String toString() {   // overrides generated toString method
    return String.format("Tuple2(%s, %s)", _1, _2);
  }
}

public class Main {
  public static void main(String[] args) {
    Tuple2<Integer, String> t = Tuple2.of(123, "hello");

    int x = 10 * t._1();  // _1() is a getter for first component

    System.out.println(t.swap());  // Tuple2(hello, 123)
  }
}
```

Memory Layout en Java

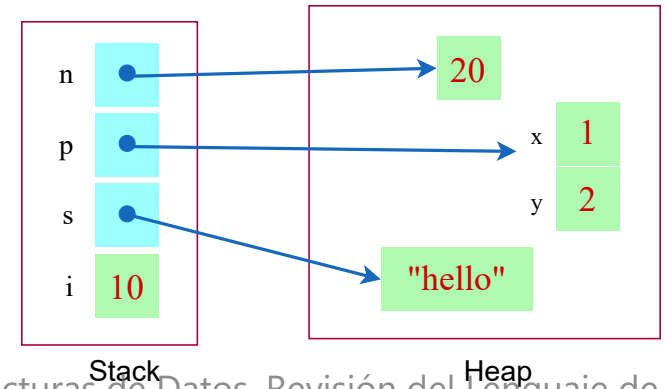
- **Contrastando tipos y objetos primitivos**
  - **Tipos primitivos**: como `int`, `double`, ... se almacenan sus valores en memoria directamente.
  - **Objetos (tipos de referencia)**: Representado por *referencias* (punteros). La referencia se almacena en un área de memoria (pila o montículo), mientras que el objeto en sí reside siempre en el mónticulo.
- **Memoria de pila** `stack`
  - Almacena variables *locales* (pueden ser tipos primitivos o referencias a objetos) y parámetros de llamada a métodos.
  - La gestión se realiza *automáticamente* a medida que los métodos son llamados y retornan.
- **Memoria del heap** `heap`
  - Almacena todos los objetos creados y sus atributos, independientemente de dónde se declaren las referencias a objetos.
  - Los objetos se *alojan* cada vez que se llama a `new`.
  - *Garbage Collected*: El programador no necesita liberar objetos. Los objetos inalcanzables son liberados periódicamente por el recolector de basura.

Ejemplo de diseño de memoria en Java.

```java
class Point {
  private int x, y;  // instance attributes

  public Point(int x, int y) {  // constructor
    this.x = x;
    this.y = y;
  }
}

public class Main {
  public static void main(String[] args) {
    int i = 10;                 // int is a primitive type, stored in stack
    String s = "hello";         // s is a reference to a String object in heap
    Point p = new Point(1, 2);  // p is a reference to a Point object in heap
    Integer n = 20;             // n is a reference to an Integer object in heap
  }
}
```



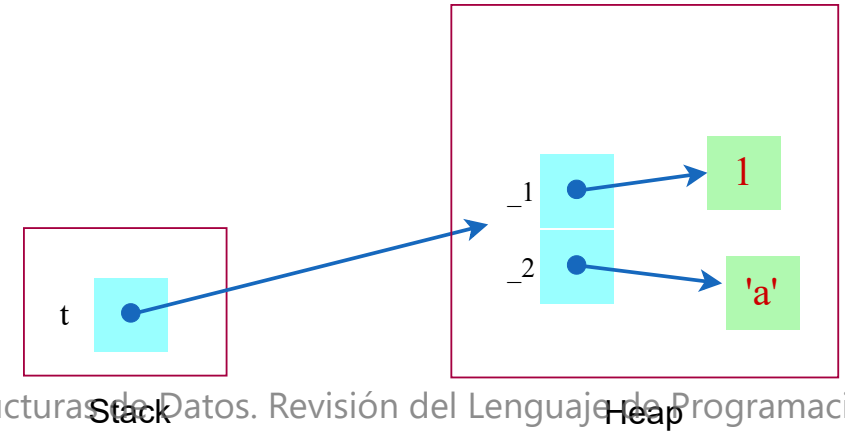
Ejemplo de uso de genéricos y diseño de memoria en Java

- Los genéricos siempre se instancian con *objetos*, por lo que se representan mediante *referencias*.

```
class Tuple2<A, B> {
  private A _1;
  private B _2;  // instance attributes

  public Tuple2(A _1, B _2) {  // constructor
    this._1 = _1;
    this._2 = _2;
  }
}

public class Main {
  public static void main(String[] args) {
    Tuple2<Integer, Character> t = new Tuple2<>(1, 'a');
  }
}
```



Simplifica las declaración de variables `var`

- ¿Qué es `var`?
  - El *compilador* determina el tipo de la variable *local* en función del valor de inicialización.
- Beneficios:
  - Más compacto:
    - Reduce la *redundancia*.
    - Mejora la *legibilidad* del código centrándose en el valor, no en el tipo.
  - Mantiene la *seguridad de tipo*:
    - El tipo *inferido* sigue estando *fuertemente tipado*.
    - No hay pérdida de *seguridad de tipo*; solo que menos verbosidad.

Ejemplo de uso de `var`

```
public record Tuple2<A, B>(A _1, B _2) {
  public static <A, B> Tuple2<A, B> of(A x, B y) {
    return new Tuple2<>(x, y);
  }

  public Tuple2<B, A> swap() {
    return Tuple2.of(_2, _1);
  }

  public String toString() {
    return String.format("Tuple2(%s, %s)", _1, _2);
  }
}

public class Main {
  public static void main(String[] args) {
    var t1 = Tuple2.of(123, "hello");     // inferred type Tuple2<Integer, String>
    var t2 = Tuple2.of("hello", false);   // inferred type Tuple2<String, Boolean>
  }
}
```

Relaciones de orden y comparadores

- Una *relación de orden* establece una forma de comparar y ordenar objetos en función de sus valores.
- Define si un objeto es *mayor que*, *igual a* o *menor que* otro.
- En Java, un `Comparator<T>` (definido en el paquete `java.util`) representa una relación de orden para objetos de tipo `T`.

```
interface Comparator<T> {
  public int compare(T x, T y);
}
```

- `compare(x, y)` devuelve:
  - Un valor negativo, si `x` es menor que `y`.
  - Un valor positivo, si `x` es mayor que `y`.
  - 0, si `x` no es ni mayor ni menor que `y`.

Ejemplo de relaciones de orden y comparadores

```
import java.util.Comparator;

public record Employee(String name, int salary) {
  // Factory method
  public static Employee of(String name, int salary) {
    return new Employee(name, salary);
  }
}

public class Main {
  public static void main(String[] args) {
    var e1 = Employee.of("Alice", 50000);
    var e2 = Employee.of("Bob", 60000);

    // Comparator for Employee objects based on salary
    var comparator = Comparator.comparingInt(Employee::salary);

    int cmp = comparator.compare(e1, e2);
    if (cmp < 0) {
      System.out.println(e1.name() + " earns less than " + e2.name());
    } else if (cmp > 0) {
      System.out.println(e1.name() + " earns more than " + e2.name());
    } else {
      System.out.println(e1.name() + " earns the same as " + e2.name());
    }
  }
}
```

Clases internas estáticas

- Una clase interna *estática* es una clase definida dentro de otra clase.
- `static` significa que no está asociado con una instancia de la clase externa.
- Se puede crear una instancia sin una instancia de la clase externa.

```
public class LinkedList<T> implements List<T> {
  private static final class Node<E> { // static inner class representing a node
    E element;
    Node<E> next;

    public Node(E element, Node<E> next) {
      this.element = element;
      this.next = next;
    }
  }

  private Node<T> first, last; // instance attributes for LinkedList
  private int size;
  // ... rest of LinkedList implementation
}
```