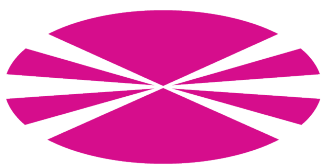
Unit 3: Basic Properties of Object Orientation

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science, Faculty of Computer Science



UNIVERSIDADE DA CORUÑA

Table of Contents

1 Abstraction and encapsulation

2 Modularity

3 Hierarchy

4 Polymorphism

5 Typing

6 Dynamic binding

Table of Contents

1 Abstraction and encapsulation
- Abstraction
- Encapsulation

2 Modularity

3 Hierarchy

4 Polymorphism

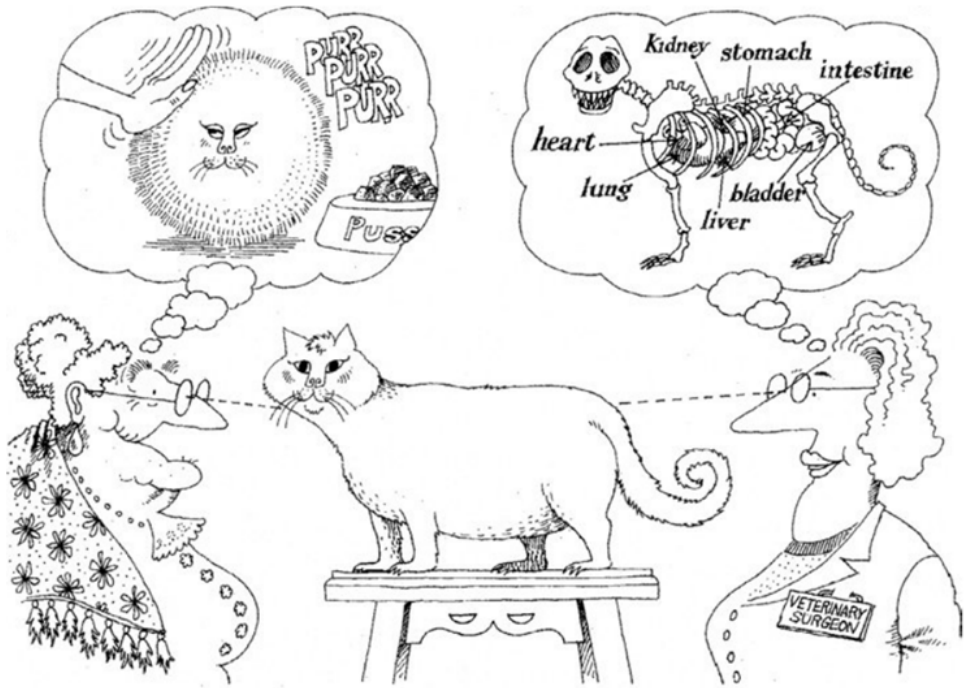
5 Typing

6 Dynamic binding

Abstraction

Abstraction

Representing the essential characteristics disregarding irrelevant details.



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

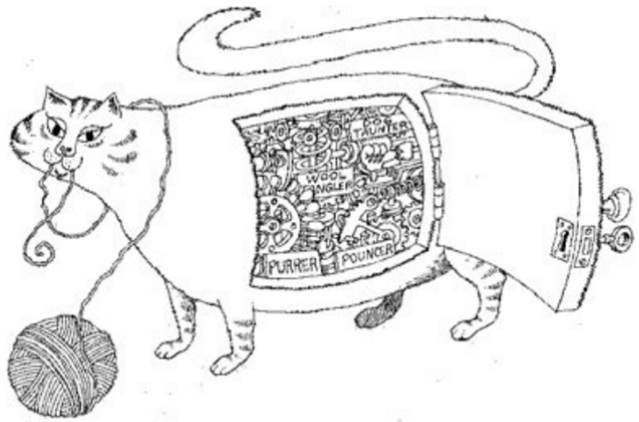
Abstraction

- One of the essential methods for tackling software's inherent complexity (as seen on ADTs).
- Abstraction's key element in OO is the concept of class.
- A class describes a set of objects by identifying their state and shared behavior.
- **Example: Sphere**
  - State: The coordinates of the center and the radius.
  - Behavior: Move center, change radius, calculate area, calculate perimeter, etc.

Encapsulation

Encapsulation

Concealing those implementation details that are not relevant when used from outside.



Encapsulation hides the details of the implementation of an object.

- Sometimes, the term is also used to describe the process of storing structure and behavior in the same container.

Abstraction vs. encapsulation

- Complementary concepts.
- Abstraction refers to observable behavior.
- Encapsulation refers to the actual implementation of that behavior (e.g., access specifiers, etc.)

Benefits

- **Concealing information**
  - Hiding low-level details lets you think about the operation or object more efficiently.
  - If you change the internal, private representation of an object (e.g., refactoring a method to optimize it) it will not negatively affect its clients.
- **Storing state and behavior together**
  - Everything about an object is there in the class $\Rightarrow$ naturalness, simplicity, etc.

Table of Contents

1 Abstraction and encapsulation

2 Modularity
- Definition and Characteristics
- Modularity in Java
- Java Packages
- Goals for the Packages

3 Hierarchy

4 Polymorphism

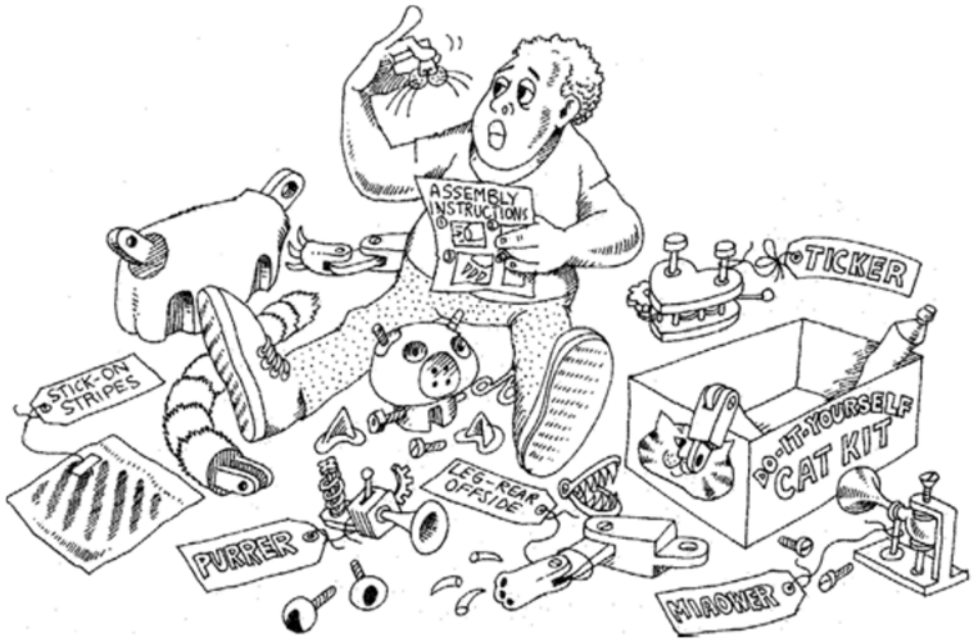
5 Typing

6 Dynamic binding

Modularity

Modularity

Breaking down a system into parts or modules that are cohesive and loosely coupled.



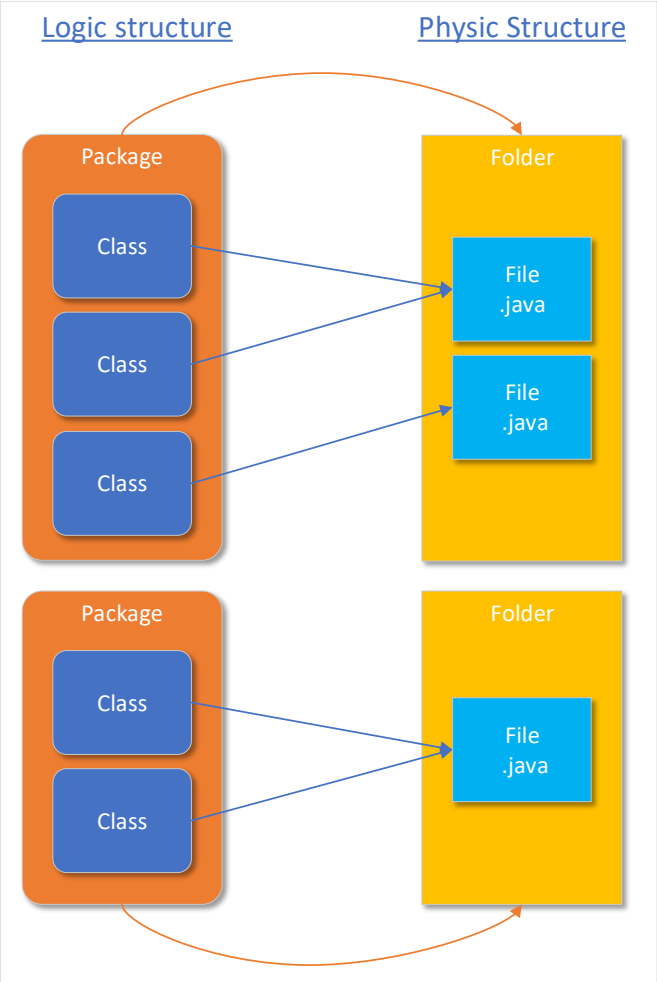
Modularity packages abstractions into discrete units.

Benefits

- **Benefits**
  - Breaking down a program into individual components lets you reduce its complexity.
  - It creates well-defined frontiers in the code $\Rightarrow$ makes it easier to understand.
- **Relationship with encapsulation**
  - Modules can hide information from other modules, thus behaving like a higher-level encapsulation mechanism.

Modularity in Java

- Modularity is implemented in two ways: **logically through classes and packages and physically through files and folders**
- **Classes**:
  - Encapsulating attributes and methods in the same container.
  - Hiding from view implementation details (by means of access specifiers).
  - This protection is highly configurable via distinct access levels.



Modularity in Java

Java files

They serve as physical compilation units.

- **Java files and classes**: A `.java` file can contain multiple classes, with the following restrictions:
  - Only one of the classes can be public.
  - The filename must be identical to the public class (if such a class exists).
- **Compiling files**
  - Compiling a `.java` file generates as many `.class` files (bytecodes) as classes exist in the file.
  - For the sake of simplicity, **it is often recommended that a file contain only one class**.

Modularity in Java

Java packages

- Logical units for grouping classes.
- Information hiding: Non-public classes can only be viewed from inside the package.

Declaring packages

- The `package` directive is placed at the beginning of the file.
- All the classes below will belong to the package.
- If no package is specified, all the classes will belong to the "default package" (not recommended!).
- Package names are in lowercase (no "Camel Case").

Java package

```
package graphics;

public class Circle {
    private int radius;

    public int getRadius() {
        return radius;
    }

    public void setRadius(int radius) {
        this.radius=radius;
    }

    //...
}
```

Importing packages

- The name of a class consists of the name of its package + a period + the name of the class (e.g., `graphics.Circle`).
- This means that whenever we want to use a class we must "always" type the name of the package.

Using classes and packages without `import`

```
graphics.Circle c1 = new graphics.Circle();
```

Importing packages

The `import` statement

Incudes the contents of the specified package into the namespace of the importing class.

- This way, we can omit the name of the package.
- Imports are place immediately below the `package` directive.

Using classes and packages with `import`

```
package otherpackage;
import graphics.Circle;

// ...

Circle c1 = new Circle();
```

Importing packages

- **Ways of importing:**
  - Importing an individual class.
  - Importing all the classes in a package.

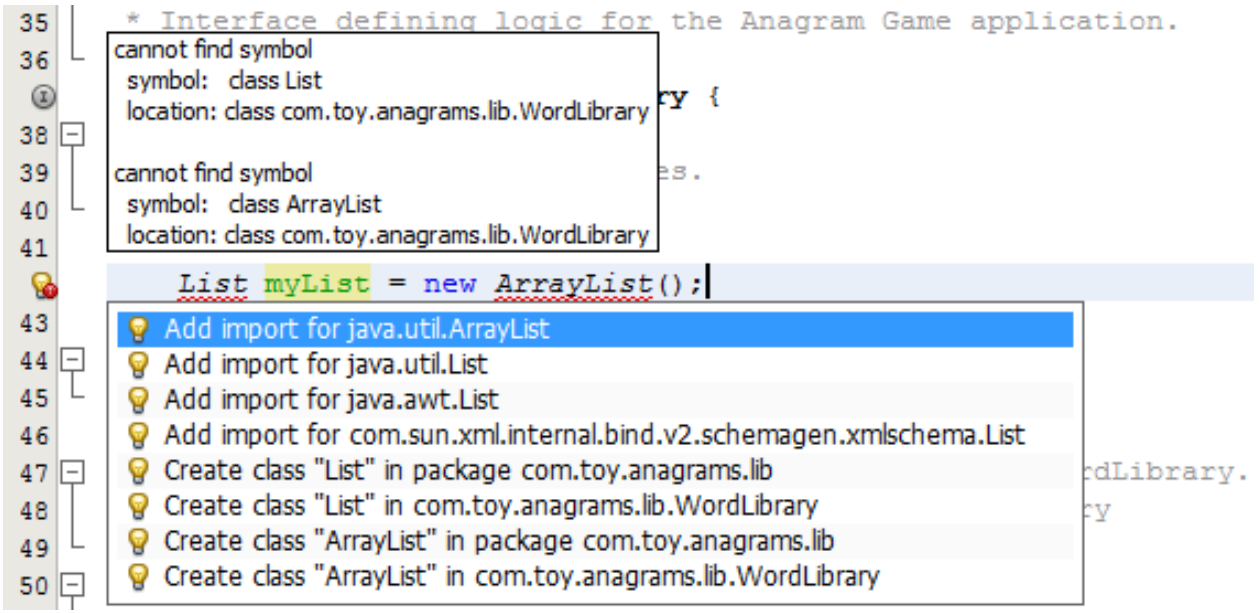
Ways of importing

```
package otherpackage;
import graphics.Circle;  // Imports only Circle
import graphics.*;       // Imports everything
```

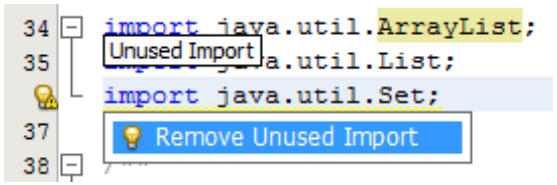
- The first way is recommended, as it clearly states what is being imported.
- The first way is more verbose, but modern IDEs facilitate the task by suggesting imports, warning about unused imports, etc.

Managing packages in NetBeans

- Suggesting imports



- Warning about unused imports



Importing packages

Careful!

The `import` statement does not grant access privileges.

- An `import` is not a `uses` from Pascal.
- Classes can be accessed by their full name anyway (i.e., `package.Class`).
- For the sake of convenience, some packages and classes from the Java API, such as `java.lang` (e.g., `Object`, etc.) can be used without importing them or stating their full name.

Package hierarchies

- Package names can be structured in levels using periods as separators.
- Example: `java.util`,`java.util.jar`, etc.

Important!

In spite of this, no actual hierarchies exist. The concept of "subpackage" does not exist in Java.

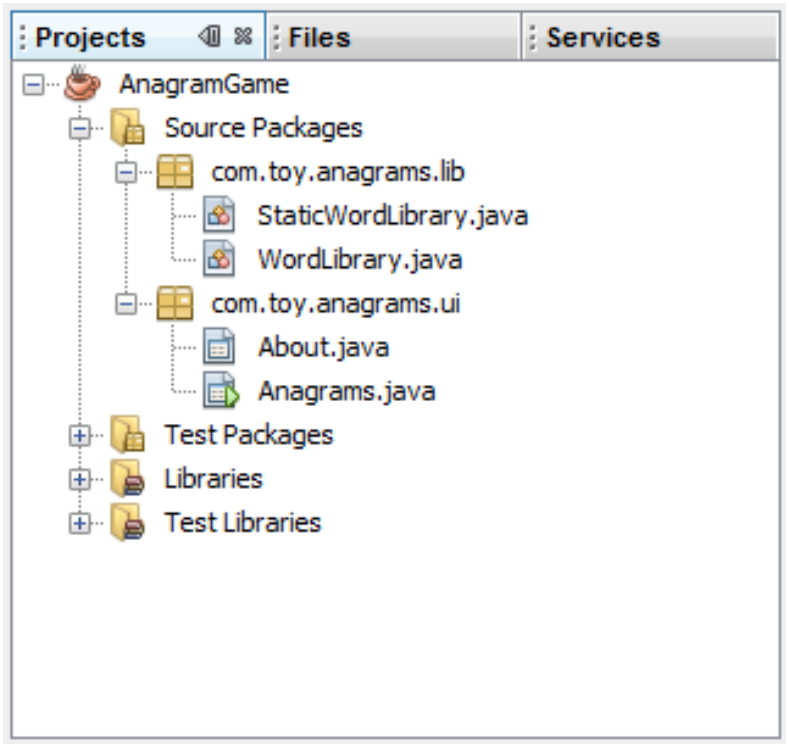
- This is simply done for clarity and organization.
- `java.util` and `java.util.jar` are actually completely different packages, just like, e.g., `java.util` and `java.swing`.

Goals for the packages

1. **Providing a high level of modularity**.
   - A package can contain both public and non-public classes.
2. **Grouping classes with similar functionalities**.
3. **Organizing source files**.
4. **Preventing naming conflicts**.

Goals for the packages

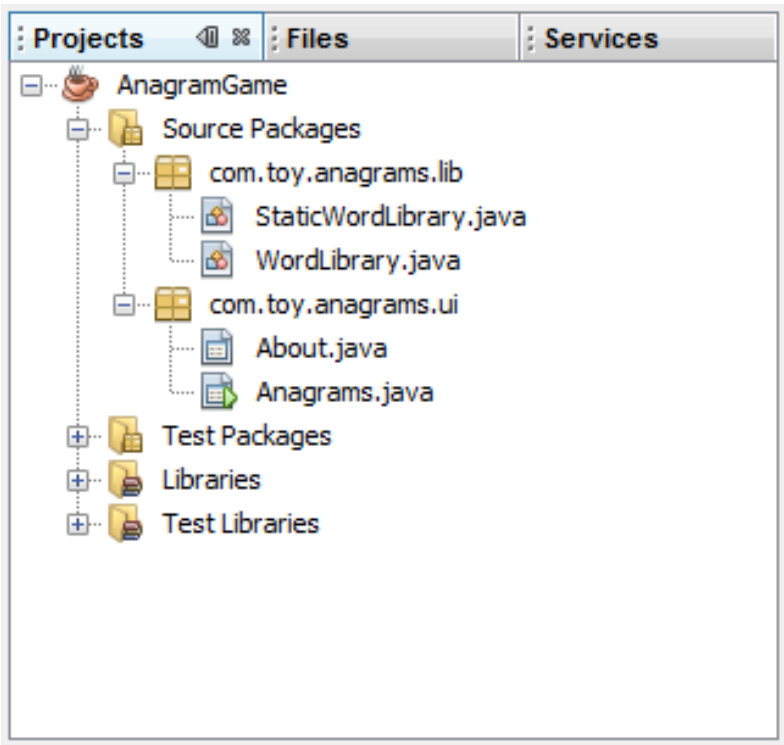
2. **Grouping classes with similar functionalities**
   - Example: User interface classes are in the `ui` package.

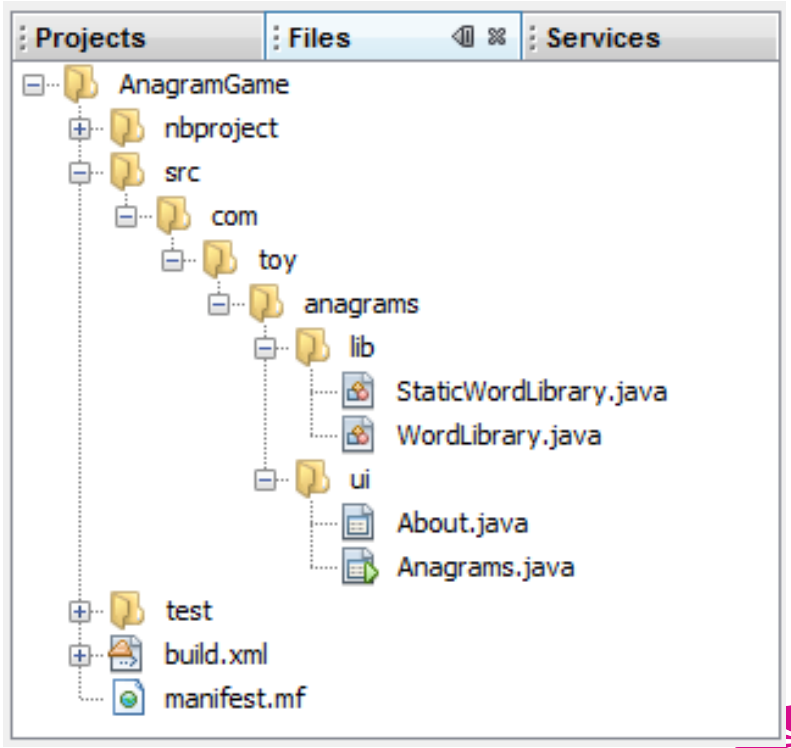


Goals for the packages

3 Organizing source files

- In Java, package names correspond to hard-drive folders.





Goals for the packages

4. **Preventing naming conflicts**
   - Packages inherently minimize naming conflicts, as the name of the package is placed before the name of the class.
   - Example: In the Java API, some class names are duplicated on completely different packages, e.g., `java.awt.List` and `java.util.List`.

- **How can we prevent naming conflicts between different packages?**
  - Internet domain names are typically used (backwards), as they are unique.
  - Example: Google packages start with `com.google`, etc.

The new Java 9 modules...

Java 9 modules are not part of this course so... **forget this slide!**.

- **Java 9 modules** add a higher level of aggregation beyond packages.
- **Reliable configuration**: Modules provide mechanisms for explicitly declaring dependencies between them.
- **Strong encapsulation**: The packages in a module are accessible to other modules only if the module explicitly exports them.
- **Scalable Java platform**: The platform is now modularized and you can create custom runtimes consisting of only the modules needed.

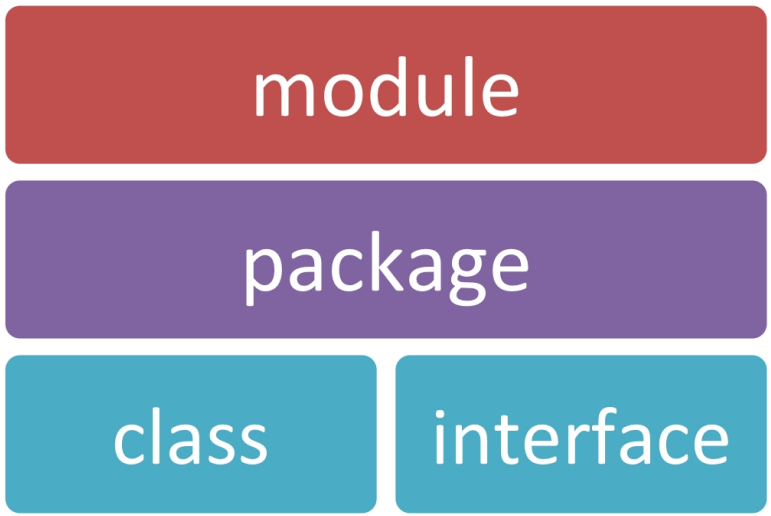


Table of Contents

1 Abstraction and encapsulation

2 Modularity

3 Hierarchy
- Composition
- Inheritance
- Abstract Classes
- Interfaces
- Inheritance vs. Composition

4 Polymorphism

5 Typing

6 Dynamic binding

Hierarchy

Hierarchy

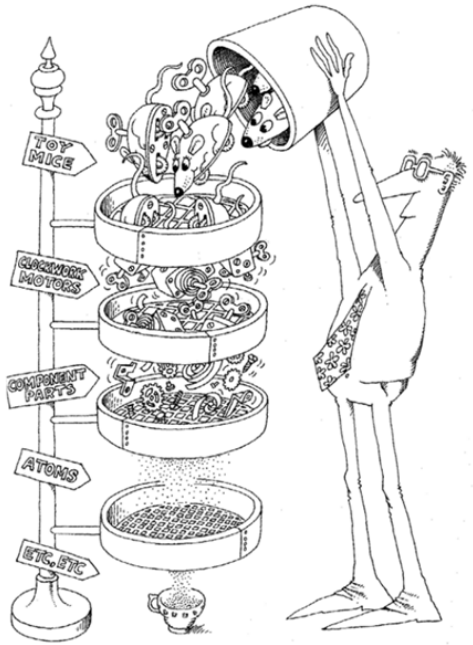
A hierarchy is a classification of abstractions.

- In OO, there are two kinds of hierarchies:
  - **Composition**.
  - **Inheritance**.

Composition

Composition

Composition defines HAS_A relations. That is, an element contains other elements (e.g., an object's state consists of other objects).



Abstractions form a hierarchy.

Composition

- The simplest kinds of objects are made up of primitive types (e.g., `int`, `float`, etc.) or objects that behave similarly to primitive types (e.g., `String`, `Date`, `BigDecimal`, etc.)

Simple object

```
public class Address {
    String street;
    int number;
    String floor;
    String city;
    int zipCode;
    String country;

    public String toString()
    {
        return number + " " + street +
               " " + floor + ", " +
               zipCode + " " +
               city + ", " + country;
    }
}
```

Composition

- Composite objects include also other objects in their state (in this example, `Address`)
- Typically, composite objects receive requests that are in turn delegated to the internal objects (e.g., `physicalAddress`) ⇒ **Delegation mechanism**.

Composite object

```
public class Person {
    String firstName;
    String lastName;
    String phone;
    String email;
    int age;
    Address address;

    public String fullName() {
        return firstName + " " + lastName;
    }

    public String physicalAddress() {
        return address.toString();
    }
}
```

Composition

Exercise: Card game

Define the class `DeckOfCards`.

- Define its state. How can you store a collection of objects?
- What would its behavior be?
- Should it be immutable?

Composition

Exercise: Card game

Delegation: Write the `toString()` method for the class `DeckOfCards`.

Inheritance

Inheritance

It's an IS_A relation between classes. One class inherits the structure and behavior of one or more classes.

Inheritance

- A class inherits from their superclasses and expands or redefines their structure and behavior.
- Subclasses represent specialized concepts.
- Superclasses represent generalizations of the common aspects of the subclasses.
- In Java, inheritance is specified using the `extends` clause at the top of the class.

Example of inheritance

- `Student` extends `Person` and inherits its attributes and methods.
- Attributes inherited from `Person`:
  - `firstName`, `lastName`, `phone`, `email`, `age` and `address`
- Specific attributes:
  - `degree` and `courses`
- Methods inherited from `Person`:
  - `fullName` and `physicalAddress`
- Specific methods:
  - `calculateTuition`

Inheritance (class `Student`)

```
public class Student extends Person {
    Degree degree;
    Course[] courses;

    public int calculateTuition() {
        // ...
    }
```

Example of inheritance

- `Professor` extends `Person` and inherits its attributes and methods.
- Attributes inherited from `Person`:
  - `firstName`, `lastName`, `phone`, `email`, `age` and `address`
- Specific attributes:
  - `department`, `professionalCategory`, `numberOfHours` and `seniority`
- Méthods inherited from `Person`:
  - `fullName` and `physicalAddress`
- Specific methods:
  - `calculateSalary`

Inheritance (class `Person`)

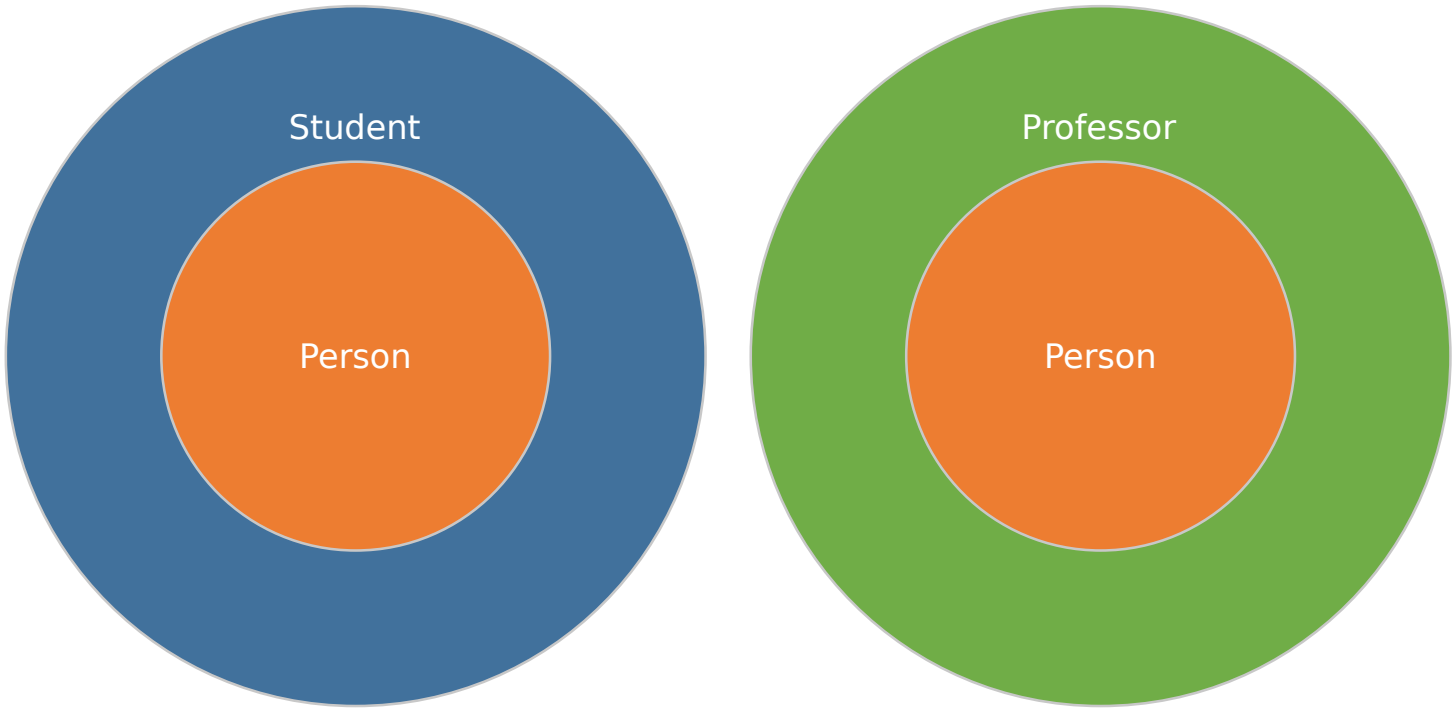
```
public class Professor extends Person {
    String departament;
    String professionalCategory;
    int numberOfHours;
    java.util.Date seniority

    public int calculateSalary() {
        // ...
    }
```

Noteworthy aspects in inheritance

- Both `Student` and `Professor` have an IS_A relation with `Person`.
- This means that both classes can be used when an object of the type `Person` is required $\Rightarrow$ **Polymorphism** (described later).
- Internally, the elements from a superclass appear in the same place in the subsequent subclasses.
  - `firstName` will be the first attribute and `fullName` will be the first method, for both classes.

Noteworthy aspects in inheritance

- An object of the subclass always contains an object of its superclass inside of it.



Inheritance structures

- Inheritance implementation ultimately depends on aspects like **single inheritance vs. multiple inheritance** and the **root classes**:

Single inheritance

A class can only have a single parent class.

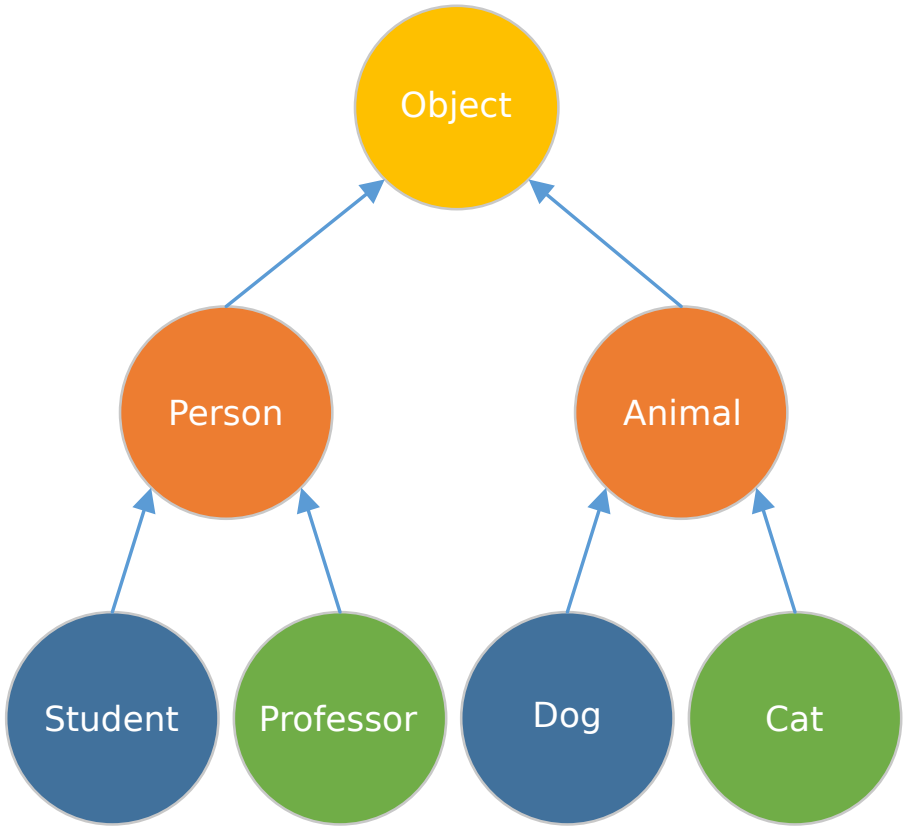
- **Languages**:
  - Java, Object Pascal, C#, etc.
- **Benefits**:
  - Inheritance structures are simple and tree-like.

Inheritance structures

Root class

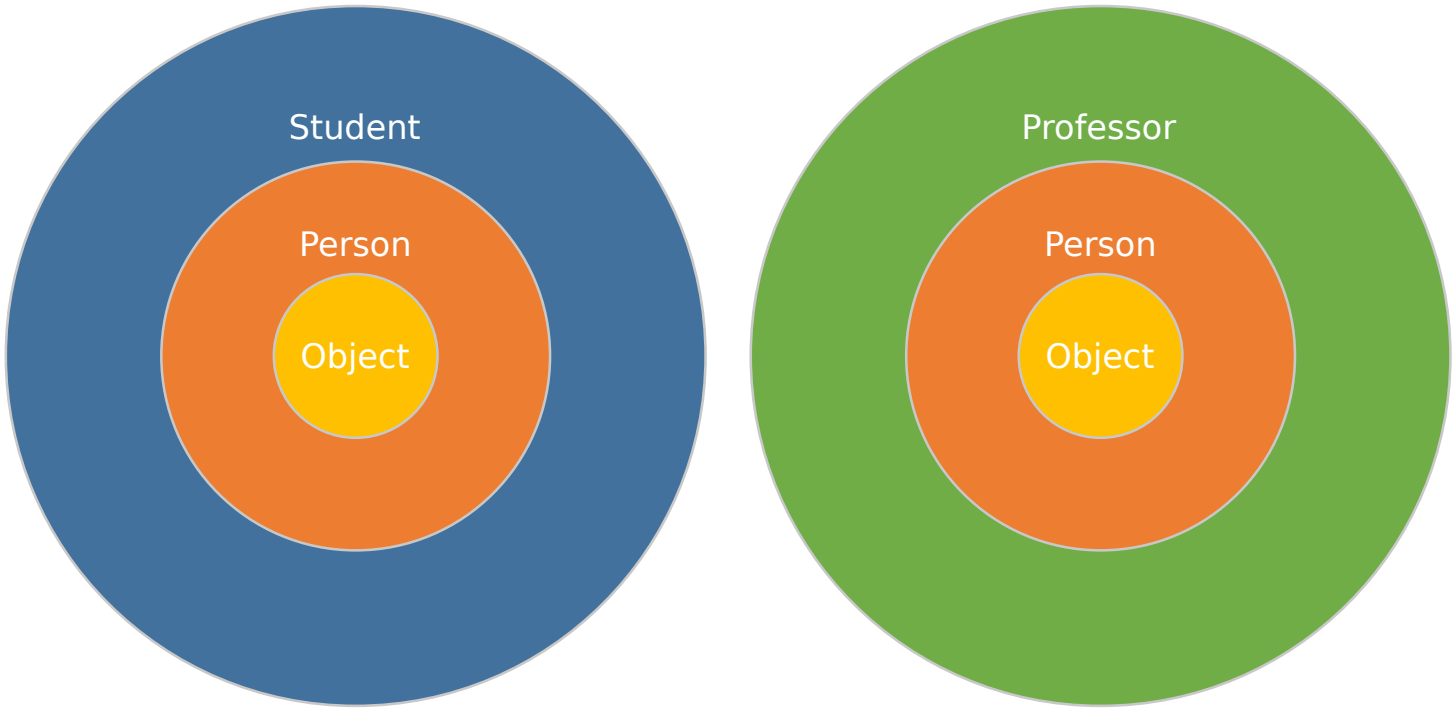
The superclass of the other classes.

- Combined with single inheritance, it generates a tree structure with a single root.
- Java is structured this way, with the class `Object` at the root.
- If a class signature does not include the `extends` clause, it is automatically a subclass of `Object`.



Inheritance structures

- Therefore, any Java object contains an instance of `Object` inside.



Inheritance structures

Multiple inheritance

A class can inherit from multiple classes at the same time.

- **Languages**:
  - C++, Eiffel, etc.
- **Benefits**:
  - More design possibilities.
- **Drawbacks**:
  - It generates conflicts whose resolution depends on the particular PL $\Rightarrow$ increases complexity and reduces understandability.
  - The consensus is to avoid multiple inheritance. Single inheritance is perfectly capable of implementing complex and flexible hierarchies $\Rightarrow$ see **Interfaces**.

Inheritance and constructors

What is the constructor of Subclass?

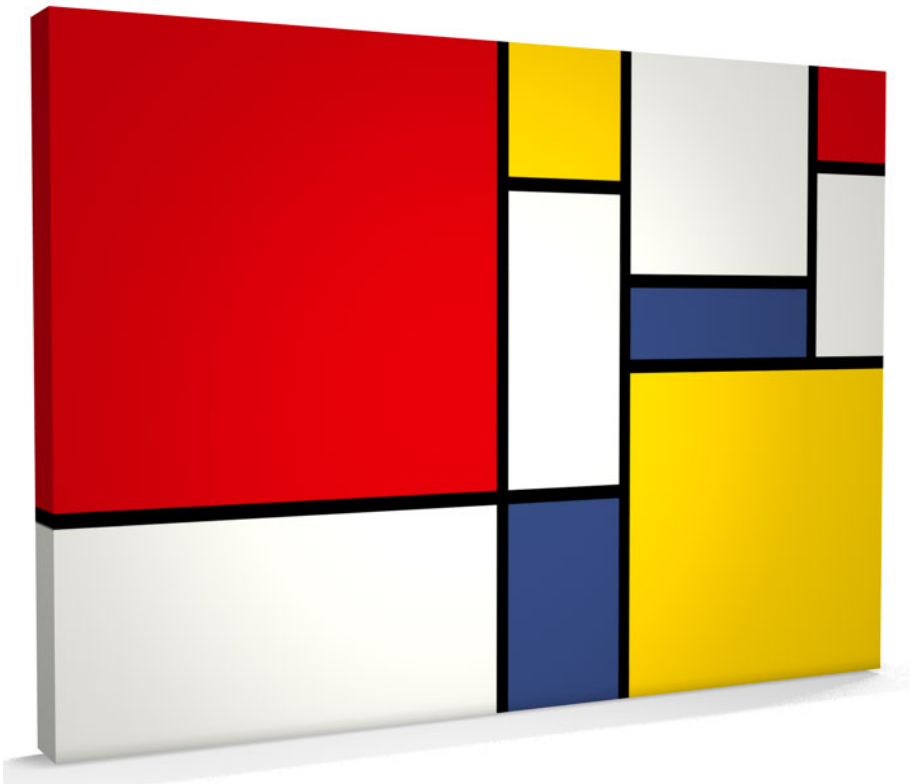
```
class SuperClass {
    private int value;

    public SuperClass(int value) {
        this.value = value;
    }
}

class SubClass extends SuperClass {
    private int subValue;
    // Constructor?
```

- An object from a subclass always contains an object from the superclass.
- This object from the superclass is created before the subclass object is instantiated.
- The `super` keyword is used to call the constructors from the superclass (analogously to how `this` was used to call another constructor from the same class).
- You can't use `super()` and `this()` in the same constructor.

Abstract classes

Abstract classes

Abstract classes cannot be instantiated (i.e., you cannot create objects of that class). They exist to be extended through inheritance.



Goals of abstract classes

1. **Defining classes that group common characteristics**
   - Classes that are made to be extended, not instantiated.
   - Example: `Animal`.
   - It makes no sense to create an instance of `Animal`. Instead of that, you instantiate subclasses like `Cat`, `Dog`, etc.

Example

```
public abstract class Animal {
    private String name;
    private String code;
    private Date birthDate;

    public getName()
    { return name; }
}

public class Cat extends Animal {
    public enum Breed
        {SIAMESE, PERSIAN};
}

public class Dog extends Animal {
    public enum Breed
        {GERMAN_SHEPHERD, BEAGLE};
}
```

Abstract methods

Abstract methods

Abstract methods just offer a public interface (i.e., name, parameters and return type). They have no implementation, which is delegated to the subclasses.

- **Important aspects:**
  - In Java, they are defined using the `abstract` modifier.
  - An abstract method cannot belong to a non-abstract class. However, an abstract class can have non-abstract methods.
  - The subclasses of an abstract class must give an implementation to the inherited abstract methods. However, these subclasses can avoid this responsibility by declaring themselves abstract.

Goals of abstract classes

2. **Delegating behavior to subclasses**
   - Some methods can be defined in the superclass at the level of public interface, but the specific implementation depends on the subclass.
   - For example, we can have an abstract `area()` method for `Figure`, but the actual implementation will depend on the type of figure (e.g., `Circle`, `Rectangle`, etc.)

Example

```
public abstract class Figure {
    public abstract double area();
    public abstract double perimeter();
}

public class Circle extends Figure {
    private double radius;
    public double area() {
        return Math.PI*radius*radius;
    }

    public double perimeter() {
        return 2*Math.PI*radius;
    }
    // ...
}
```

Goals of abstract classes

3. **Maintaining consistency in the public interface of the subclasses**
   - Subclasses must implement all the inherited abstract methods. This ensures that all subclasses will offer those methods and they will all have the expected signature.
   - For example, as `Rectangle` extends `Figure`, the compiler forces the former to implement the `area()` and `perimeter()` methods.

Example

```
public abstract class Figure {
    public abstract double area();
    public abstract double perimeter();
}

class Rectangle extends Figure {
    private double base;
    private double height;
    public double area() {
        return base*height;
    }

    public double perimeter() {
        return (base*2)+(height*2);
    }
    // ...
}
```

Abstract classes

Abstract classes can include non-abstract elements

```
public abstract class Figure {
    private int x; // Coordinates
    private int y;

    public void moveTo(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public abstract double area();
    public abstract double perimeter();
}
```

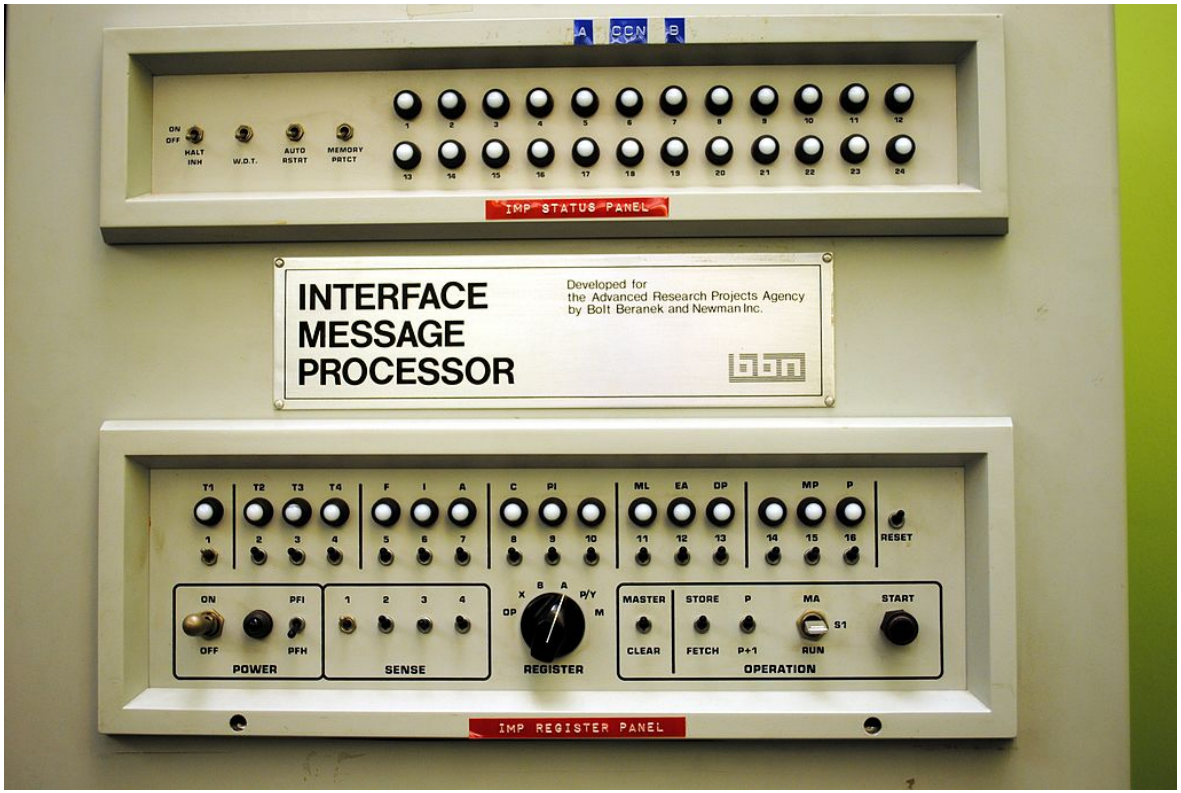
`moveTo` is common to all figures

Likewise for `getX` and `getY`

Interfaces

Interfaces

Interfaces are a sort of "purely" abstract classes that do not include *any* implementations.



Interfaces in Java

- They are defined using the keyword `interface`, instead of `class`.
- They only offer signatures for their methods, which are implicitly *public* and *abstract*.
- Interfaces can contain attributes, but they will be implicitly *public*, *static* and *final*. That is, they are class constants.

Interfaces in Java

Declaring an interface

```
[public] interface Name [extends superinterface, ...]
{ // Static constants // Abstract methods  }
```

Example

```
public interface Figure {
    double area();      // Implicitly public and abstract
    double perimeter(); // Implicitly public and abstract
}
```

- It looks like interfaces are very similar to abstract classes with abstract methods...
- ...and then we lose the possibility of including attributes and non-abstract methods. **What's the point of interfaces, then?**

Inheritance between interfaces

Inheritance between interfaces

An interface can inherit from other interfaces (including multiple inheritance!) but not from other classes.

Example

```
interface Monster {
    void threaten();
}
interface DangerousMonster extends Monster {
    void destroy();
}
interface Lethal {
    void kill();
}
interface Vampire extends DangerousMonster,
                                        Lethal {
    void drinkBlood();
}
```

Inheritance between interfaces

- Multiple inheritance is not a problem here, as interfaces only include the methods' signatures. Therefore, there is no risk of inheriting conflicting implementations.
- In the previous example, the interface `Vampire` would provide the following methods: `threaten`, `destroy`, `kill` and `drinkBlood`.
- Interface inheritance is analogous to class inheritance.

Implementing interfaces

Implementing interfaces

Implementing an interface consists in writing code for all its methods.

Example

```
public interface Figure {
    double area();
    double perimeter();
}

class Circle implements Figure {
    private double radius;
    public double area() {
        return Math.PI*radius*radius;
    }
    public double perimeter() {
        return 2*Math.PI*radius;
    }
}
```

- The "implementation" relation is indicated by the keyword `implements`.
- A class that implements an interface is forced to implement all its methods (e.g., `area()` and `perimeter()`).

Implementing interfaces

- A class can implement multiple interfaces.

Example

```
class Virus implements Monster, Lethal {
    public void threaten() {
        // Contagion
    }

    public void kill() {
        // Death by infection
    }
}
```

Implementing interfaces

- A class can avoid the responsibility of implementing methods by declaring itself abstract.

Example

```
abstract class AbstractVirus implements Monster, Lethal {
    // Does not implement any methods
}

class Ebola extends AbstractVirus {
    public void threaten() {
      // Non-abstract class are obliged to provide implementations
    }

    public void kill() {
      // ...
    }
}
```

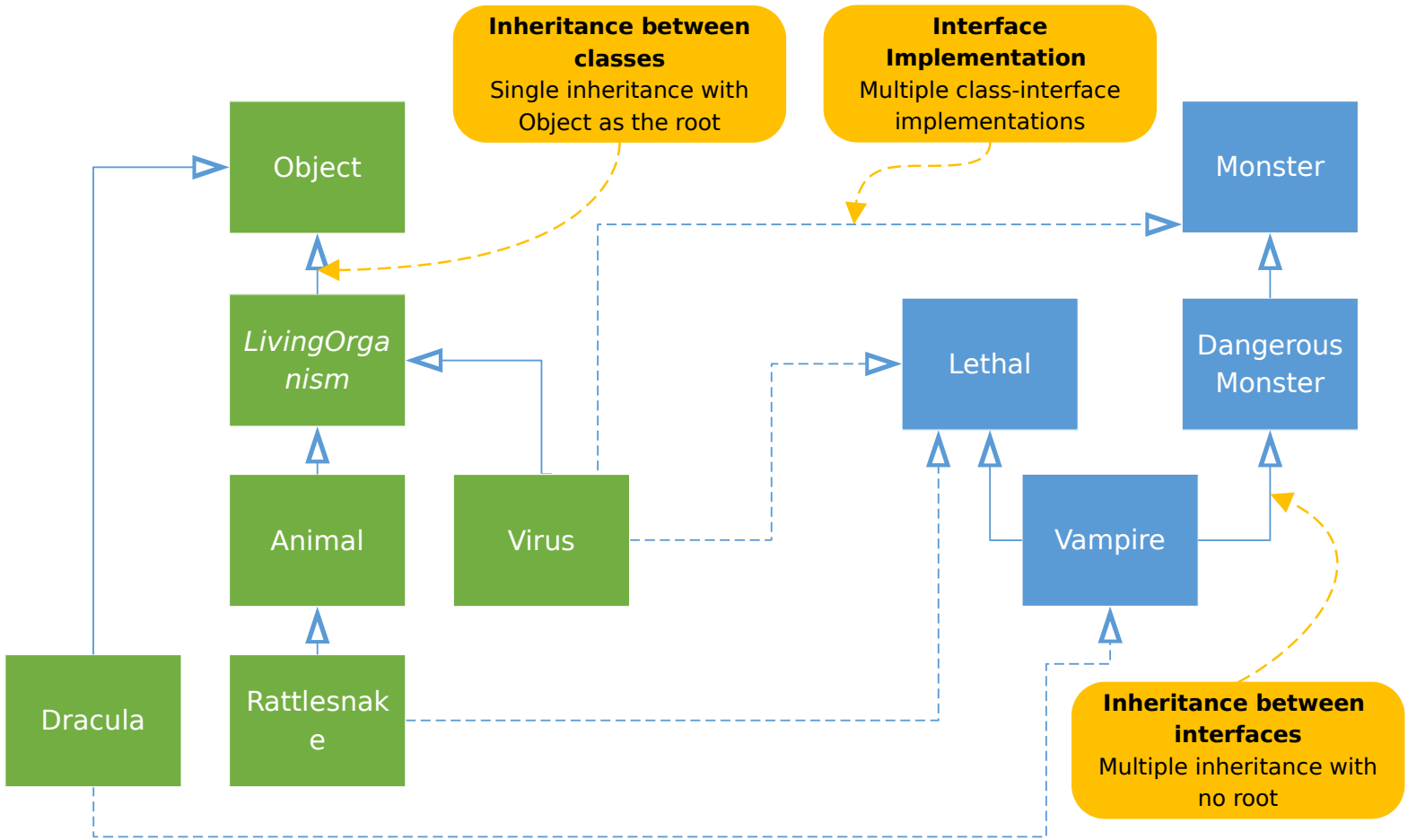
Implementing interfaces

- We could also provide a "placeholder" implementation with the idea of coding the real implementation later.
- Careful! This can be dangerous ⇒ see **Unit 5: Limitation inheritance**

Example

```
class Godzilla implements DangerousMonster {
    public void threaten() {
    throw new UnsupportedOperationException("Not supported yet.");
    }

    public void destroy() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

Hierarchies of classes and interfaces



Exercise

Exercise: Card game

Even though our simple class game does not need inheritance, we discussed the possibility of including it on unit 2. How would you do it?

- Would you use abstract classes or interfaces?
- Which classes would be abstract classes/interfaces and which ones would be concrete classes?
- Which methods would be abstract? Which methods would be specific to a subclass/implementing class?

Using interfaces

- **Common uses of interfaces**
  1. Disclosing the public interface without disclosing the actual class.
  2. Capturing the similarities in unrelated classes.
  3. Defining new data types.

- **Uses of interfaces that are not recommended**
  4. Marker interfaces.
  5. Containers for global elements.

Common uses of interfaces

1. **Disclosing the public interface without disclosing the actual class**
   - If we do not reveal the actual classes in the internal implementation, these classes can be refactored without negatively affecting its clients.
   - **Example**: Suppose we define an interface for sorting arrays.

Interface `Sorting`

```
public interface Sorting {
    void sort(int[] array);
}
```

Disclosing the public interface without disclosing the actual class

- We can now define implementing classes, with their specific characteristics, benefits and drawbacks.

Implementing classes

```
public class BubbleSorting implements Sorting {
    public void sort(int[] array) {
        // ...
    }
}

public class QuicksortSorting implements Sorting {
    public void sort(int[] array) {
        // ...
    }
}
```

Disclosing the public interface without disclosing the actual class

- Client classes are unaware of the actual implementation of the underlying algorithm, which can be dynamically replaced.

Implementing classes

```
public class Client {
    private int[] array;
    public Client(int[] array) { this.array = array; }
    public void sortArray(Sorting algorithm) {
        algorithm.sort(array);
    }
    public static void main(String[] args) {
        int[] array1 = {5, 9, 8, 4, 1, 3, 7, 2, 6};
        int[] array2 = {3, 5, 1, 7, 8, 6, 2, 4, 9};

        Client c1 = new Client(array1);
        c1.sortArray(new BubbleSorting());

        Client c2 = new Client(array2);
        c2.sortArray(new QuiksortSorting());
    }
```

Common uses of interfaces

2. **Capturing the similarities in unrelated classes**
   - Classes that are unrelated by inheritance can still be considered instances of the same class if they implement the same interface.
   - **Example**: Suppose we are developing an adventure game and we create an interface for every action (e.g., fight, swim, fly, etc.)

Interfaces representing actions

```
public interface CanFight { void fight(); }

public interface CanSwim { void swim(); }

public interface CanFly { void fly(); }
```

Capturing the similarities in unrelated classes

- Heroes can carry out some of the actions. Monsters can carry out others, etc.

Class `Hero`

```
public class Hero implements
             CanFight, CanSwim, CanFly {

  public void fight() {
    System.out.println("A hero can fight");
  }

  public void swim() {
    System.out.println("A hero can swim");
  }

  public void fly() {
    System.out.println("A hero can fly");
  }
}
```

Class `Monster`

```
public class Monster implements
             CanFight, CanFly {

  public void fight() {
    System.out.println("Monster fight");
  }

  public void fly() {
    System.out.println("Monster fly");
  }
}
```

Capturing the similarities in unrelated classes

- A class such as `Adventure` can treat both heroes and monsters as instances of the same class (`CanFight`)
- **If a class implements an interface, we consider it an instance of that interface.**

Class `Adventure`

```
public class Adventure {
    public static void fight(CanFight x) { x.fight(); }
    public static void swim(CanSwim x) { x.swim(); }
    public static void fly(CanFly x) { x.fly(); }

    public static void main(String[] args) {
        Hero johnMcClane = new Hero();
        Monster godzilla = new Monster();

        if (johnMcClane instanceof CanFight)
            fight(johnMcClane);
        if (godzilla instanceof CanFight)
            fight(godzilla);
    }
}
```

Common uses of interfaces

3 Defining new data types

- An interface is a new data type.
- At any point, we can add new instances of said data type. We only need to define new classes that implement the interface.
- **Example**: The class `Scanner` from the Java API lets you do syntactic analyses using regular expressions.
- The constructor in `Scanner` accepts objects from the classes `String`, `File` and `InputStream`. But it also accepts any object that implements the interface `Readable`.

Class `Scanner`

```
public class Scanner {
    public Scanner(String source) { ... };
    public Scanner(File source) throws FileNotFoundException { ... }
    public Scanner(InputStream source) { ... }
    public Scanner(Readable source) { ... }
    // ...
}
```

Defining new data types

- The interface `Readable` represents objects that are a source of characters.
- These characters are read by the method `read` and stored in a `CharBuffer`.
- The method returns an integer with the number of characters stored in the buffer, or a `-1` if it reaches the end of the source.

Interface `Readable`

```
public interface Readable {
    public int read(CharBuffer cb) throws IOException;
}
```

Defining new data types

- Any class can now implement the interface `Readable` and be used by `Scanner` seamlessly.
- For example, the `RandomIP` class creates 4 random numbers in an IP address.

Class `RandomIP`

```
public class RandomIP implements Readable {
    private int count = 4;
    private static Random rand = new Random();

    public int read(CharBuffer cb) {
        if (count-- == 0) { return -1; }
        String result = Integer.toString(rand.nextInt(255)) + " ";
        cb.append(result);
        return result.length();
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(new RandomIP());
        while (s.hasNextInt()) { System.out.print(s.nextInt() + " "); }
    }   // Scanner uses the read() method  from the interface Readable.
}
```

Uses of interfaces that are NOT recommended

4 **Marker interfaces**

- Marker interfaces contain neither attributes nor methods.
- Therefore, their implementing classes inherit nothing and are under no obligation to implement anything.
- However, they are *marked* by said interface (the dependencies between class and interface can be consulted with the `instanceof` operator).

Example

```
public class CloneableClass implements Cloneable {
    // It has no obligations whatsoever

    public static void main(String[] args) {
        CloneableClass cc = new CloneableClass();
        if (cc instanceof Cloneable) {
            // We check if it is marked as Cloneable.
        }
    }
}
```

Annotations vs. marker interfaces

- We argue that marker interfaces are not a good use of interfaces because, since Java version 5, there is a better mechanism for marking ⇒ **annotations**.

Annotations

Annotations are metadata. They are similar to javadoc tags (e.g., `@author`) but applied to code.

- The Java API includes a number of annotations (e.g., `@Override`, `@Deprecated`, `@SupressWarnings`, etc.) but their versatility lies in allowing the creation of custom annotations ⇒ Used profusely in libraries like JUnit, etc.
- Unlike marker interfaces, annotations allow to annotate methods and attributes, in addition to classes.

The `@Deprecated` annotation

- It marks a method as *deprecated* so the compiler advises against using it.
- It is different from the annotation of the same name used in Javadoc (which, surprisingly, the compiler also used to read and still does – due to backwards compatibility).
- It is recommended to use both annotations (one for the comments and the other for compilation).

The `@Deprecated` annotation

```
public class DeprecatedClass {
    /**
     * Does something...
     * @param value over which it does something
     * @deprecated Used the doSomething method instead
     */
    @Deprecated public void doesSomething(int value) { }
    // ...
}
```

Custom annotations

Example

We store the percentage of progress in an int. The RUNTIME parameter means that the annotation can be read at runtime.

We mark this class as "under construction" and indicate a percentage.

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@interface UnderConstruction { int percentage(); }

@UnderConstruction (percentage=50)
class MyClass {  }

public class SimpleAnnotation  {
    public static void main(String[] args) {
        MyClass x = new MyClass();
        Class c = x.getClass();
        if (c.isAnnotationPresent(UnderConstruction.class)) {
            System.out.println("This class is under construction");
            UnderConstruction ed =
                (UnderConstruction)c.getAnnotation(UnderConstruction.class);
            System.out.println("Percentage = " + ed.percentage());
        }
    }
}
```

The percentage is read at runtime

Uses of interfaces that are NOT recommended

5 **Containers for global elements**

- Java does not support global variables, which leads some programmers to use interfaces as containers for constants.

Example

```
public interface Constants {
    double PI = 3.1415926;
    int MAX_ROWS = 65536;
}

class Circle implements Constants {
    private double radius;

    public double area() {
        // We inherit PI from Constants
        return PI*radius*radius;
    }
    // ...
}
```

Interfaces as containers for global elements

- **Problems:**
  - The list of interfaces that a class implements is openly public. We are conspicuously disclosing implementation details $\Rightarrow$ **Polution of the exported API**.
  - We are creating a dependency between `Circle` and `Constants` that, being public, is also difficult to change.
- **Solutions:**
  - Define the constants as enumerations.
  - Place the constants in related classes (`MIN_VALUE` and `MAX_VALUE` are in the `Integer` class).
  - Create utility classes to store the constants (the `Math` class stores mathematical constants such as `PI` or `E`).

Interfaces as containers for global elements

Utility classes as containers for constants

```
public final class Math {
    public static final double E = 2.71828182845904523536;
    public static final double PI = 3.14159265358979323846;

    // ...
}

class Circle {
    private double radius;

    public double area() {
        return Math.PI*radius*radius;
    }

    public double perimeter() {
        return 2*Math.PI*radius;
    }
    // ...
}
```

Note that we are typing the name of the class before the constant (as it is static).

Interfaces as containers for global elements

- **Solution:**
  - Using *static imports*.
  - They are similar to class imports, but they import static elements (attributes and methods).

Utility classes as containers for constants

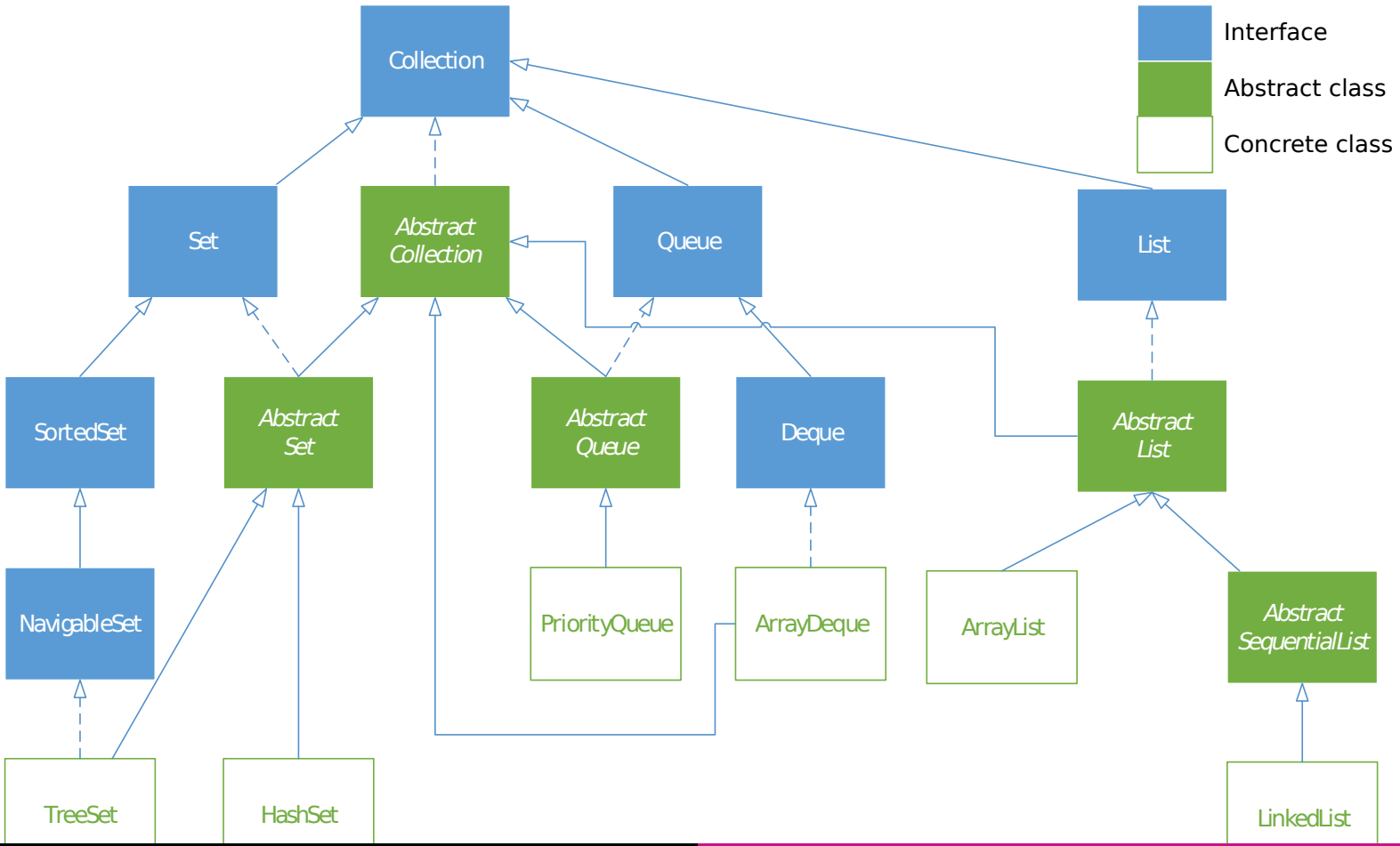
```
import static java.lang.Math.PI; // Imports only the PI constant.
import static java.lang.Math.*;  // Imports all static elements in Math.

class Circle {
    private double radius;

    public double area() {
        return PI*radius*radius;
    }
    public double perimeter() {
        return 2*PI*radius;
    }
    // ...
}
```

Interfaces as containers for global elements

- **Be careful with:**
  - Overusing static imports. The imported elements may look like they belong to the importing class, but they actually belong to another $\Rightarrow$ causes confusion.
  - The "*" notation. Instead, it is generally preferable to import individual elements.

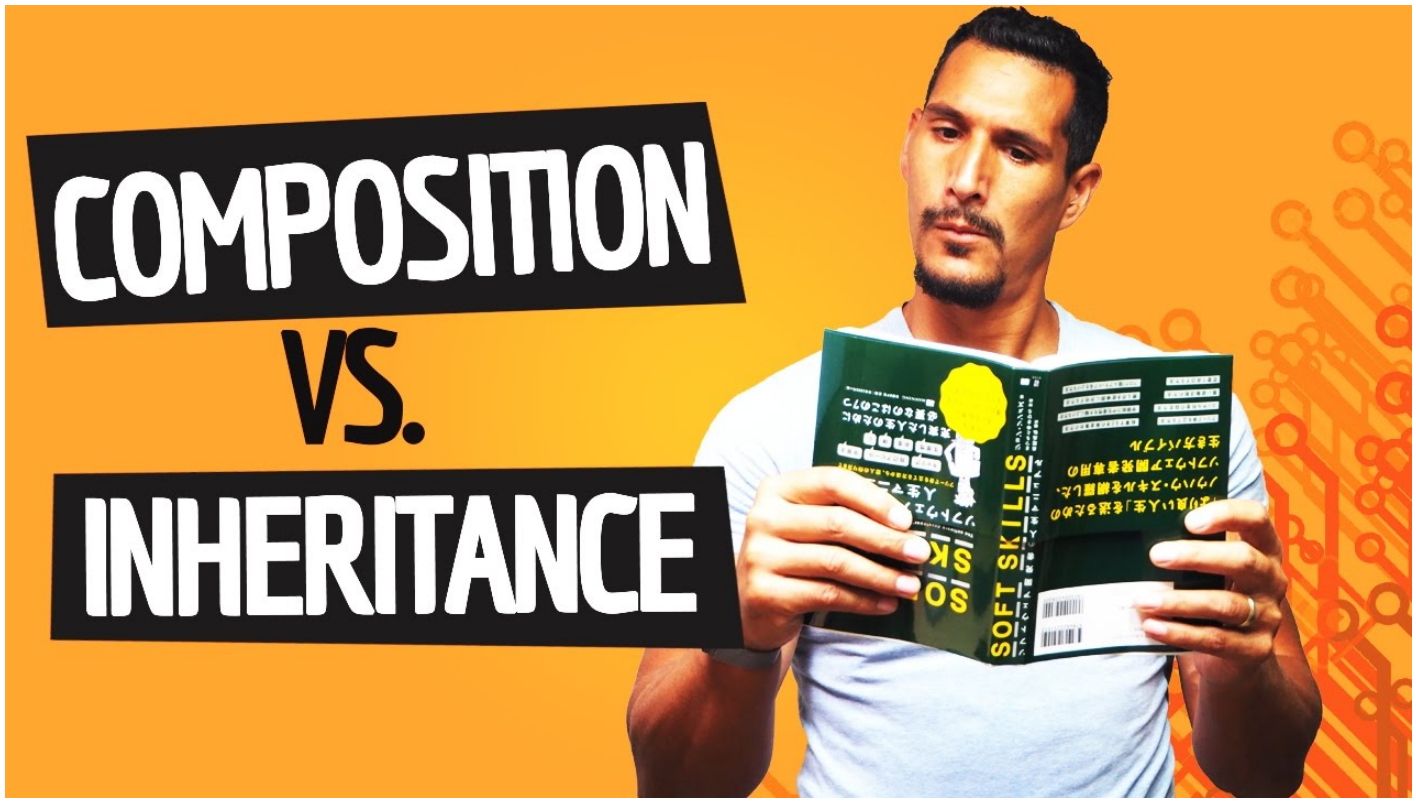
Interfaces or abstract classes?

- The collections API from Java uses both. They complement each other.



Inheritance vs. composition

- Both inheritance and composition are reusability mechanisms for classes.
- **What should we use? Inheritance or composition?**



Inheritance vs. composition

- `Student` would use *delegation* to call the methods in `Person`.
- **Do you think this is a good approach?**

Example

```
public class Student {
    Degree degree;
    Course [] courses;
    private Person p;

    public String fullName() {
        return p.fullName();
    }

    public String physicalAddress() {
        return p.physicalAddress();
    }
    public int calculateTuition() {
        // ...
    }

    //...
}
```

Inheritance vs. composition

- There is a design principle that states "**Favor composition over inheritance**".
- Why, if it's seemingly less convenient, as you need to write delegation methods?

Inheritance vs. composition

- **Counterexample**: The Java API defined the class `Stack` as extending the class `Vector`.
- `Vector` is an array-based list (similar to `ArrayList`).

Example

```
public class Stack extends Vector {
  public Object push (Object item) {
       addElement(item);
       return item;
  }
  public synchronized Object pop() {
       int len = size();
       Object obj = peek();
       removeElementAt(len - 1);
       return obj;
  }
  public synchronized Object peek() {
       int len = size();
       if (len == 0)
          throw new EmptyStackException();
       return elementAt(len - 1);
  }
  public boolean empty() {
       return size() == 0;
  }
}
```

What's wrong with the `Stack` class?

Using composition instead

- Now, there is no confusion over which methods exist.
- Encapsulation is maintained (you can only remove the element at the top).
- You cannot pass a `Stack` where a `Vector` is required.
- It is easy to replace `Vector` with a newer class such as `Arraylist`, as private elements do not affect anything outside the class.

`Stack` implemented with composition

```
class Stack {
  private Vector data;
  public Stack () {data = new Vector();}
  public boolean isEmpty () {
      return data.isEmpty();
  }
  public Object push (Object item) {
      data.addElement (item);
      return item;
  }
  public Object peek () {
      return data.lastElement();
  }
  public Object pop () {
      Object obj = data.lastElement();
      data.removeElementAt
                        (data.size()-1);
      return obj;
  }
}
```

Benefits of inheritance

- **Inheritance implicitly supports the substitution principle**
  - An instance of a subclass is also an instance of its superclass, and can be passed where an object of the superclass is required.
- **Coding is quicker**
  - When you inherit methods and attributes, you can use them directly. There is no delegation.
- **Inheritance grants access to protected elements**
  - The `protected` access specifier was specifically created for this purpose. (However, remember that in Java protected elements are also visible from the same package).
- **Relations between classes are more public**
  - Inheritance is openly disclosed in the public interface of the class.

Benefits of composition

- **Composition states clearly which operations can be used.**
  - Inheritance is more opaque because you need to climb up and down the inheritance tree to understand how a class works.
  - With composition, you can refuse to inherit undesirable methods (e.g., `removeElementAt` in `Stack`).
- **Composition makes refactoring easier**
  - Internal private objects can be changed without affecting the client classes.
- **Composition does not imply substitution**
  - Composition is a HAS_A relation, not an IS_A relation.
- **Composition is a more general solution**
  - Inheritance has limitations (e.g., single vs. multiple inheritance) that composition has not (e.g., we can use composition with several classes simultaneously).

Inheritance vs. composition

Conclusions

- Use inheritance for those relations that are clearly of the IS_A type. That is, those in which it makes sense to pass an object of the subclass where a superclass is needed.
- Use composition for other types of relations.
- When in doubt, prioritize composition over inheritance.

Table of Contents

1 Abstraction and encapsulation

2 Modularity

3 Hierarchy

4 Polymorphism
- Types of Polymorphism
- Overloading
- Coercion
- Inclusion Polymorphism
- Parametric Polymorphism

5 Typing

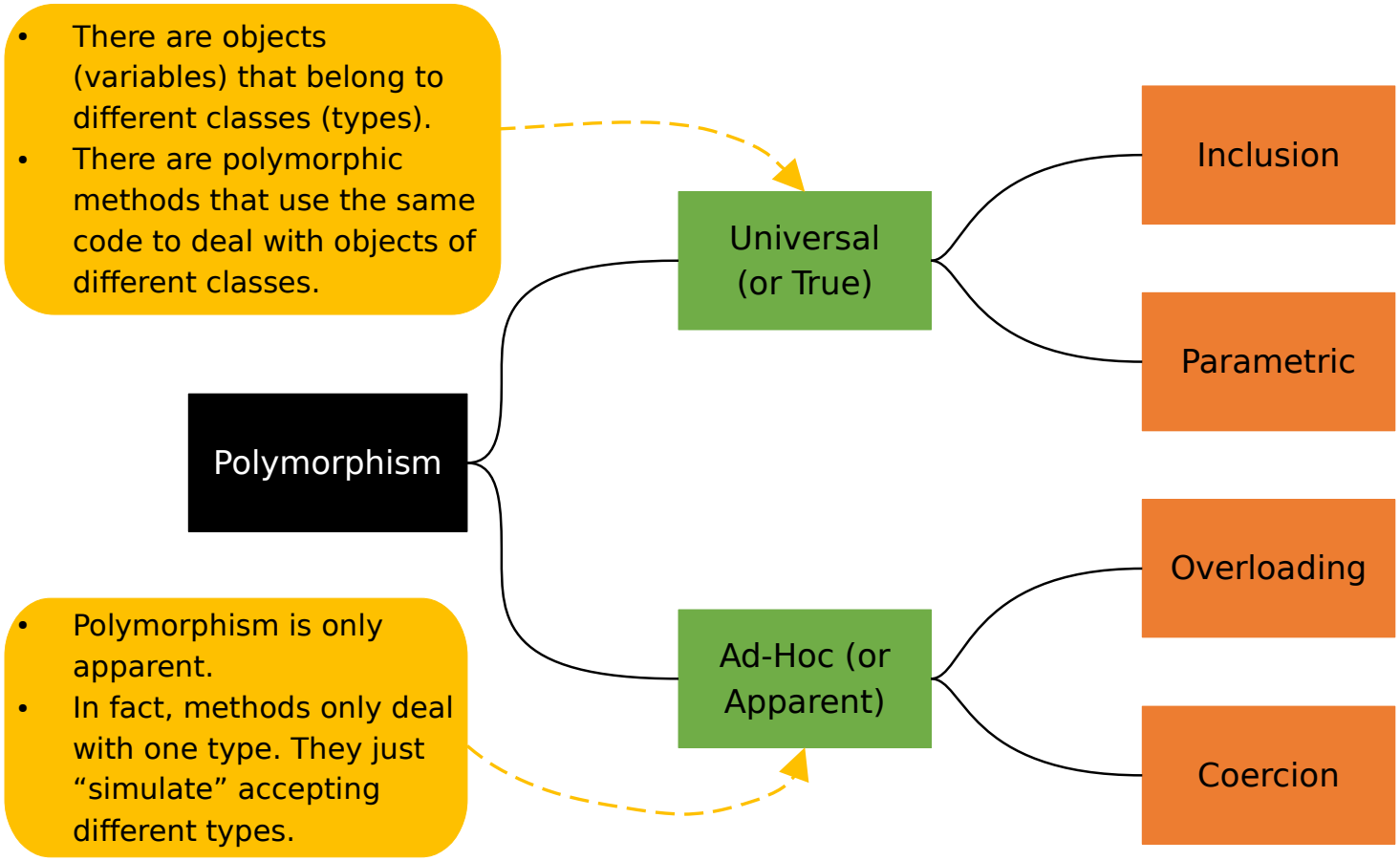
6 Dynamic binding

Polymorphism

Polymorphism

When an object belongs to more than one class, or a function is applied to a parameter from different classes.

Cardelli and Wegner's classification



Overloading

Overloading

Using the same name for methods that are ultimately different. The methods are differentiated by the number of parameters.

Overloading

Classical example: Overloading in constructor methods

```
public class Box {
    private int side;
    private int value;

    // Constructors
    public Box() { this(0, 10); }

    public Box(int value) { this(value, 10); }

    public Box(Box b) { this(b.value, b.side);  }

    public Box(int value, int side) {
        this.value = value;
        this.side = side;
    }
}
```

Overloading

Overloading in other methods

```
public class Overloading {
    public void methodX (String s) {
        System.out.println("String " + s);
    }

    public void methodX (int i) {
        System.out.println("Integer " + i);
    }

    public void methodX (int i, int j) {
        System.out.println("Integers " + i + " and " + j);
    }

    public void methodX (int i, String s) {
        System.out.println("Integer " + i + " and string " + s);
    }
}
```

This polymorphism is merely apparent. This is *not* a single method that accepts a variety of parameter types but several methods with the same name.

Overloading and return types

Is this code valid?

```
public class Overloading {
    public int methodX() {
            return 1; // Returns an int
    }

    public double methodX() {
            return 1.0; // Returns a double
    }
}
```

Class-level overloading

Parameter-based overloading

We have a single class, and overloading consists in having methods with the same name and a different number or type of parameters.

Inter-class overloading

We have several classes with the same methods and the same parameters. The difference is in the objects over which they are invoked.

- **Inter-class overloading** is used to facilitate the learning of libraries that share analogous operations.
- Example: the `isEmpty` method is applicable to `String`, `List`, `Set`, etc.

Overloading between classes and subclasses

How many "methodX" are there in ChildClass?

```
public class ParentClass {
    public void methodX (int i, int j) {
        System.out.println("Integers " + i + " and " + j);
    }
    // ...
}

public class ChildClass extends ParentClass {
    public void methodX (String s) {
        System.out.println("String " + s);
    }
}
```

Overloading between classes and subclasses

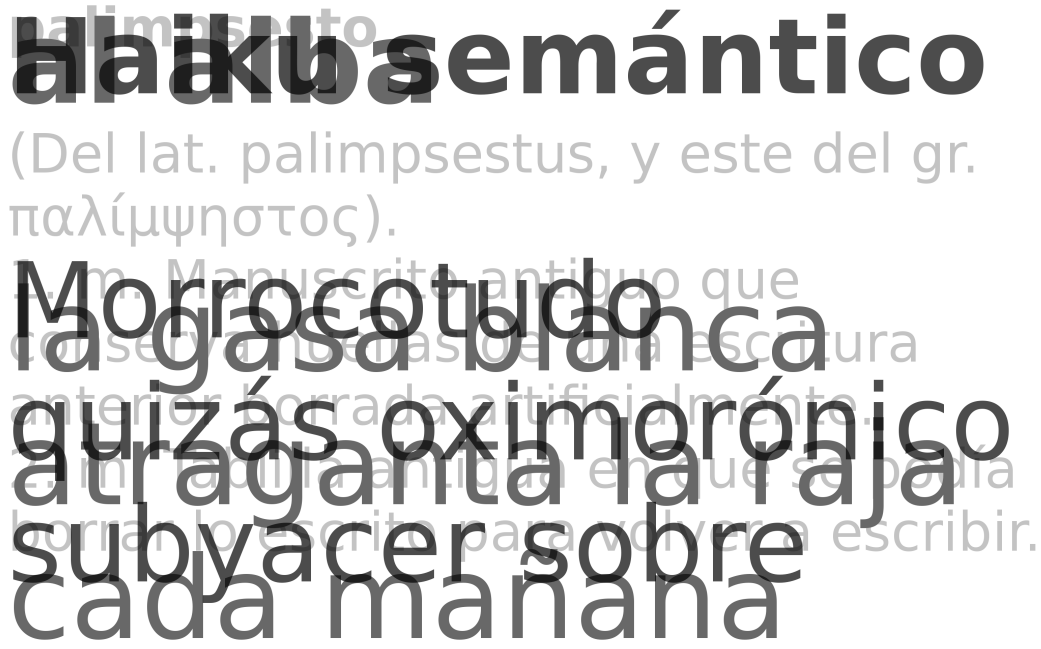
What is the result of this code?

```
public class ParentClass {
    public int methodX(int i, int j)  {
        return i + j;
    }
}

class ChildClass extends ParentClass {
    public int methodX(int i, int j)  {
        return i * j;
    }

    public static void main(String[] args) {
        ChildClass ch = new ChildClass();
        System.out.println(ch.methodX(5, 2));
    }                                    // 7, 10 or compilation error?
}
```

Overriding

Overriding

When a child class gives a more specific implementation to a method it inherits from its parent class.



Overriding vs. Overloading

- Overriding can initially seem similar to inter-class overloading (same method and parameters, different classes).
- But the inheritance that exists between them means that the subclass is *redefining* the method.
- This redefined implementation is prioritized, as it is more suited to the instances of the subclass.

Overriding

Cylinder gives a more specific implementation to `area`

```
public class Circle {
    protected double radius;

    public double area() {
        return Math.PI*radius*radius;
    }

    public double perimeter() {
        return 2*Math.PI*radius;
    }
}

public class Cylinder extends Circle {
    protected double height;

    public double area() { // Surface of the cylinder
        double aux = 2*Math.PI*radius;
        return (aux * radius) + (aux * height);
    }
}
```

Overriding

- The original methods of the superclass can still be called from the subclass if you use the keyword `super` ⇒ **Refinement overriding**.
- `super` is a pointer to the superclass (analogously to how `this` is a pointer to the current class).

Cylinder implements `area` using `super`

```
public class Cylinder extends Circle {
    protected double height;

    public double area() {  // Surface of the cylinder
      return 2*super.area() + super.perimeter()*height;
    }
}
```

Variants in overriding

- In order to override effectively, the signature of both methods must be exactly the same (name, parameters, etc.)
- **Allowed variants**:
  - The subclass method can return a object that is a **subclass** of what the superclass method returns.
  - The subclass method can change the visibility, but only to make it **more permissive** (`private` ⇒ `[package]` ⇒ `protected` ⇒ `public`)

Example of the two variants

```
class Variants {
    protected Animal getAnimal() { return new Animal(); }
}
class SubVariants extends Variants {
    public Dog getAnimal() { return new Dog(); }
}
class Animal {}
class Dog extends Animal {}
```

Overriding in Java

Are we effectively overriding `equals`?

```
public class Box {
    int value;
    public void setValue(int v) { value = v; }
    public int getValue() { return value; }

    public boolean equals(Box box) {
        if (box == null) { return false; }
        if (this.value != caja.value) { return false; }
        return true;
    }
}
```

Overriding in Java

- In Java, it is not necessary to *explicitly* state that you intend to override a method.
- **Problem**
  - If you make mistakes like typing the wrong name, the wrong parameter order, and so on, you are probably not actually overriding the method.
  - The compiler accepts your code and the problem may remain hidden.
- **Solution**
  - Use the optional `@Override` annotation to explicitly state your intention of overriding. If you make a mistake and are not actually overriding the method, compilation will fail.

Overriding in Java

Using `@Override` to check incorrect overriding

```
public class Box {
    int value;
    public void setValue(int v) { value = v; }
    public int getValue() { return value; }

    @Override
    public boolean equals(Box box) {
        if (box == null) { return false; }
        if (this.value != caja.value) { return false; }
        return true;
    }
}
```

Compilation error

Coercion

Coercion

When an argument is converted to the expected type to prevent typing errors.

Coercion

- Coercion can be performed statically – during compilation – as in *type casts*. Or it can also be performed dynamically, at runtime.
- **Example**: A function that receives a `double` as a parameter can easily accept an `int`. It will simply convert that `int` to a `double` at runtime.
- Many PLs have coercion mechanisms with clearly defined rules for type conversion.

Coercion

Example

```
public class Coercion {
    public static double multiplyByTwo(double i) {
        return i*2;
    }

    public static void main(String[] args) {
      System.out.println(Coercion.multiplyByTwo(5.5)); // Prints 11.0
      System.out.println(Coercion.multiplyByTwo(5.0)); // Prints 10.0

      // Coercion of int 5 to double 5.0
      System.out.println(Coercion.multiplyByTwo(5));   // Prints 10.0
    }
}
```

Coercion

- In Java, the conversion chain for primitive types is: `byte` ⇒ `short` ⇒ `int` ⇒ `long` ⇒ `float` ⇒ `double`.
- These types can be converted to a subsequent compatible type with no loss of information.
- A more advanced coercion mechanism exists, and it's called **Autoboxing**. It will be discussed in the next chapter (Inclusion polymorphism).

Inclusion polymorphism

Inclusion polymorphism (or subtype polymorphism)

Inheritance-based polymorphism where an instance of a subclass is also an instance of the superclass. It is typical of OOPLs.

Inclusion polymorphism

- Inclusion polymorphism makes it possible to use an instance of a subclass when an instance of its superclass is required (e.g., assignments, parameters, etc.)
- Note that the opposite is not true!

Example: `Dog` and `Cat` are subclasses of `Animal`

```
public static void method(Animal a) { ... }
// ...
Animal a1 = new Dog(); // Allowed
Animal a2 = new Cat(); // Allowed
Dog d1 = new Dog();
method(d1); // Allowed
// ...
Dog d2 = new Animal(); // Compilation error. An Animal is
                       // not necessarily a Dog
```

Inclusion polymorphism

What is the result of running this code?

```
class Polymorphism {
    public static void main(String[] args) {
        Dog d = new Dog(); // Dog extends Animal
        if (d instanceof Dog) System.out.println("d is a Dog");
        if (d instanceof Animal) System.out.println("d is an Animal");

        Animal a = new Dog();
        if (a instanceof Dog) System.out.println("a is a Dog");
        if (a instanceof Animal) System.out.println("a is an Animal");
    }
}
```

Universal methods

- Many OOPLs, such as Java, have a superclass for all classes (in this case, `Object`).
- Using `Object`, it is easy to create "universal" methods that receive parameters of any type or return values of any type.

Example: Universal method

```
public Object method(Object obj)  {
    // ...
}
```

Universal collections

Universal collection of objects: `ArrayList` (simplified version)

```
public class ArrayList {
    private Object[] elementData; // Data stored in an Object array
    private int size=0;           // Size
    public ArrayList() {
        datos = new Object[10];  // Initial capacity of 10
    }
    public boolean add(Object o) {
        elementData[size++]=o; // Not checking the end of the array
        return true;
    }
    public Object get(int index) {
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException();
        return elementData[index];
    }
    public boolean isEmpty() {
        return size == 0;
    }
    // ...
}
```

Universal collections

- It is easy to use the `ArrayList` class to create collections of dogs or cats.

Using universal collections

```
// ...
ArrayList dogs = new ArrayList();
dogs.insert(new Dog("Snowy"));
dogs.insert(new Dog("Snoopy"));

ArrayList cats = new ArrayList();
cats.insert(new Cat("Garfield"));
cats.insert(new Cat("Heathcliff"));
// ...
```

Universal collections

- **Problems with inclusion polymorphism and universal collections**
  1. You cannot create collections of primitive data types.
  2. Objects "lose their character" when they are inserted into the collection.
  3. Unsafe typing.

Universal collections

1. **You cannot create collections of primitive data types**
   - For efficiency reasons, primitive data types (`int`, `float`, etc.) are not objects in Java.
   - Collections based on inclusion polymorphism can store any subclass of `Object`, but not primitive types.

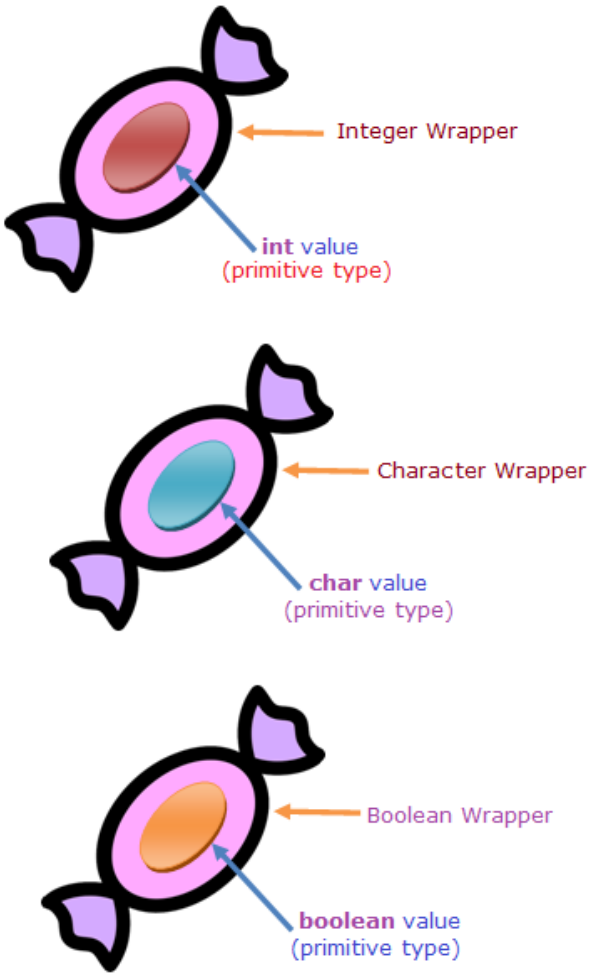
Wrapper classes

Classes that "wrap" a primitive type and provide the Object-like facade that is required in our universal collections.

- They are **immutable** and contain a private attribute of the wrapped primitive type, as well as getters and setters.
- Moreover, they include utility methods (e.g., conversions), constants, etc.

Wrapper classes

Primitive type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character



Wrapper classes

Boxing

Process of introducing a primitive type in its corresponding wrapper class.

Unboxing

Process of extracting a primitive type from its corresponding wrapper class.

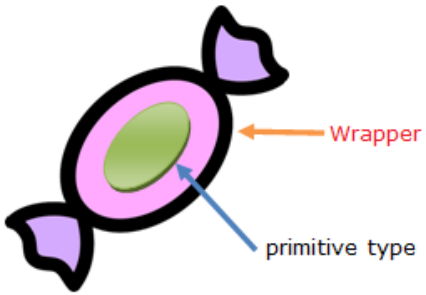


fig : Boxing

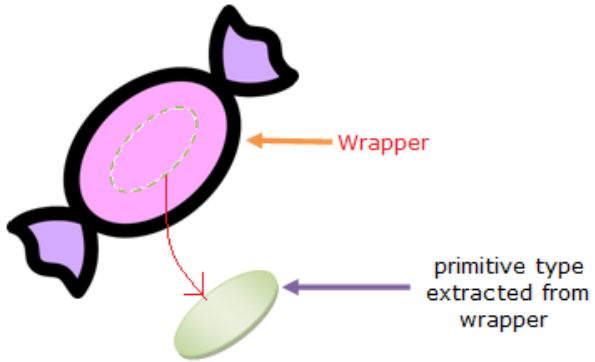


fig : Un-Boxing

Wrapper classes

Wrapper class `Integer` (simplified version)

```
public final class Integer extends Number {
    private final int value; // Internal value
    public Integer(int value) { this.value = value; } // Constructor
    public int intValue() { return value; } // Returns primitive type

    // Constants
    public static final int MIN_VALUE = 0x80000000;
    public static final int MAX_VALUE = 0x7fffffff;

    // Utility methods
    // Convert to double
    public double doubleValue() { return (double)value; }
    // Convert to String
    public String toString() { return toString(value);  }
    // Convert to String (static)
    public static String toString(int i) { ...
    }
    //...
}
```

Wrapper classes

- **Problem with wrapper classes**: they are cumbersome to use.

Example

```
ArrayList l = new ArrayList();

l.add(new Integer(1));
l.add(new Integer(2));
l.add(new Integer(3));

Integer i = (Integer)l.get(0);
System.out.printf("The value of 0 is %d", i.intValue());
// ...
```

- **Solution**: autoboxing/autounboxing.

Wrapper classes

- **Important**: Since Java 9, `Integer`'s constructor is *deprecated*.
- The static method `Integer.valueOf(int)` is a better choice.
- This method is likely to yield significantly better space and time performance by caching frequently requested values (at least the range -128 to 127).

Ejemplo

```
public final class Integer extends Number {
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    } // ...
}
// Example
ArrayList l = new ArrayList();
l.add(Integer.valueOf(1));
l.add(Integer.valueOf(2));
l.add(Integer.valueOf(3));
```

IntegerCache is a cache of Integer prepared by the JVM.

No new objects are created, it uses those in the cache.

Wrapper classes

Autoboxing

When an object is required and, as a result, a primitive type is automatically converted to an instance of its wrapper class.

Autounboxing

When a primitive type is required and, as a result, an instance of a wrapper class is automatically converted to a primitive type.

- Available since Java version 5.
- It can be considered an advanced type of Coercion.

Wrapper classes

Example of Autoboxing/Autounboxing

```
// ...
ArrayList l = new ArrayList();

l.add(1);
l.add(2);
l.add(3);

Integer i = (Integer)l.get(0);
System.out.printf("The value of 0 is %d", i);
// ...
```

- **Why do we still need an explicit type cast (i.e., `(Integer)l`) to take an object from the collection?**

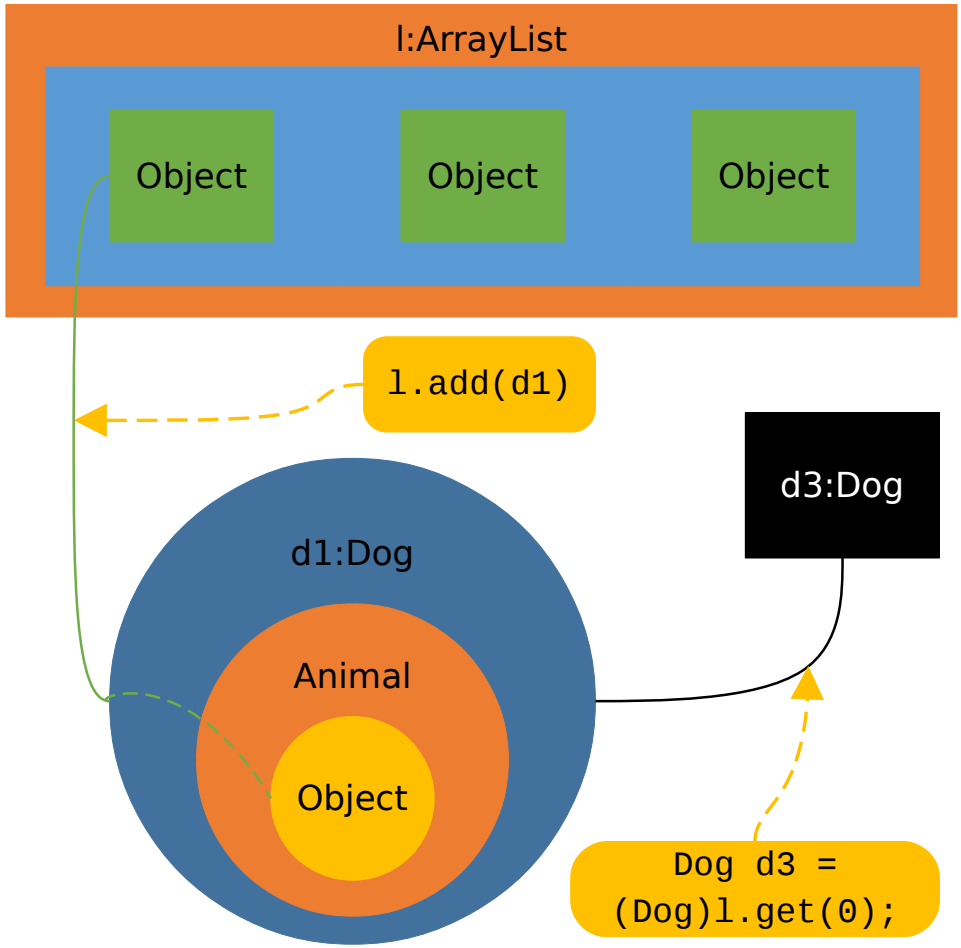
Universal collections

2. **Objects "lose their character" when they are inserted into the collection**
   - Any instance of a subclass can be used as if it were an instance of its superclass, but when you do that you lose access to the specific characteristics of the subclass.
   - However, the object is still ultimately an instance of the subclass.
   - In order to access the specific characteristics of the subclass, it is necessary to do an explicit type cast.
   - Any explicit type cast will be accepted at compile time. If the conversion is not valid, an exception will be thrown at runtime.

Loss of character

Example

```
ArrayList l = new ArrayList();
Dog d1 = new Dog("Snoopy");
l.add(d1);

// Compilation error
Dog d2 = l.get(0);
// error: incompatible types
// required: Dog found: Object

// Explicit type cast
Dog d3 = (Dog)l.get(0);
// Solves compilation errors
// May generate runtime errors

// Runtime error
Cat c1 = (Cat)l.get(0);
// Dog cannot be cast to Cat
```



Universal collections

3 Unsafe typing

- If a collection consists of `Objects`, this means that anything can be inserted into it.
- That is, nothing can stop us from inserting a cat into a collection of dogs.
- Problems arise when you extract this cat and type cast it to a `Dog`.
- The solution to those last two problems is **Generics**.

Example

```
ArrayList dogs = new ArrayList();
dogs.add(new Dog("Snowy"));     // Correct
dogs.add(new Dog("Snoopy"));    // Correct
dogs.add(new Cat("Garfield")); // Correct!!

Dog d1 = (Dog)dogs.get(0); // Correct
Dog d2 = (Dog)dogs.get(1); // Correct
Dog d3 = (Dog)dogs.get(2); // Runtime error!!
```

Universal collections

Exercise: Card game

Write the class `DeckOfCards` using an `ArrayList` collection.

Parametric polymorphism

Parametric polymorphism (generics)

When the same function is applied to different data types. The specific type at each point is indicated by an extra parameter.

Parametric polymorphism

- **Generics**
  - Generic classes
  - Type parameters
  - Instantiating generic classes
- **Generics and collections**
  - Generic collections
  - Benefits
  - `for-each` loop
- **Bounded types**
- **Wildcards**
- **Generics and subclasses**
  - Covariance
  - Parameters and subclasses
  - Wildcars and the *Get and Put Principle*
- **Generic methods**
- **Interfaces and generics**
- **Complexities of generics**
  - Backwards compatibility (*Erasure*)
  - The `unchecked` *warning*

Generics - Generic classes

- Remember the class `Box`, which stored integers.

`Box` without generics, stores `int` values

```
public class Box {
    private int value;

    public void setValue (int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public Box(int value) {
        this.value = value;
    }
    public static void main(String[] args) {
        Box b = new Box(5);
    }
}
```

Generics - Generic classes

- If we wanted to define a new type of box that stores `double` decimal numbers, we would need to create a separate new class.

`Box` without generics, stores `double` values

```
public class BoxOfDoubles {
    private double value;

    public void setValue (double value) {
        this.value = value;
    }
    public double getValue() {
        return value;
    }
    public BoxOfDoubles(double value) {
        this.value = value;
    }
    public static void main(String[] args) {
        BoxOfDoubles b = new BoxOfDoubles(5.0);
    }
}
```

Generics - Generic classes

- Before generics the solution was the use of `Object`.

Generic `Box` but using `Object`

```
public class GenericBox {
    private Object value;

    public void setValue (Object value) {
        this.value = value;
    }
    public Object getValue () {
        return value;
    }
    public GenericBox(Object value) {
        this.value = value;
    }
    public static void main(String[] args) {
        GenericBox b = new GenericBox(new Dog());
    }
}
```

Generics - Generic classes

- **Generic class**: Class that includes a parametrized type (`T`) that represents an unspecified object whose real type will be specified when creating instances of that class.

`Box` with generics

```
public class GenericBox<T> {
    private T value;

    public void setValue (T value) {
        this.value = value;
    }
    public T getValue () {
        return value;
    }
    public GenericBox(T value) {
        this.value = value;
    }
}
```

Generics - Type parameters

Type parameters

- They are generic parameters that represent an unspecified type.
- `T` is just an identifier, a sort of placeholder for a data type.
- You could use any identifier, but Java's style guide recommends using one letter: `T` for any type, `E` for an element, `K` for a key, and `V` for a value.
- Do not write a class called `T`. You just have to replace the `T` placeholder with an actually existing class when you call `new`.

Generics - Instantiating generic classes

- **Instantiating generic classes:**
  - The desired type is indicated when you instantiate a collection, via the `new` keyword.
  - Primitive types are not directly supported and can only be used through wrapper classes and *autoboxing*.

Creating instances of `Box` with generics (Java 5)

```
public static void main(String[] args) {
    GenericBox<Integer> b1 = new GenericBox<Integer>(5);
    GenericBox<Double> b2 = new GenericBox<Double>(5.0);
}
```

Generics - Instantiating generic classes

- **Diamond operator (since Java 7):**
  - Since Java version 7, you can use the *diamond operator* (<>) to infer the type from the declaration, so you don't have to type it twice.

Creating instances of `Box` with generics and diamond operator (Java 7)

```
public static void main(String[] args) {
    GenericBox<Integer> c1 = new GenericBox<>(5);
    GenericBox<Double> c2 = new GenericBox<>(5.0);
}
```

Generics - Instantiating generic classes

- **Local Variable Type Inference (since Java 10):**
  - Since Java version 10, you can use type inference in local variables.
  - The variable is defined with the `var` keyword and the compiler infers the appropriate type from the variable initializer.

Creating instances of `Box` with generics and local variable type inference (Java 10)

```
public static void main(String[] args) {
    var b1 = new GenericBox<Integer>(5);
    var b2 = new GenericBox<Double>(5.0);
}
```

Collections and generics

Class `ArrayList` with generics (simplified version)

```
public class ArrayList<E> {
    private Object[] elementData; // The array is defined using Object bc
    private int size=0;           // it must be instantiated using Object
    public ArrayList() {
        elementData = new Object[10]; // You cannot use E here
    }
    public boolean add(E element) { // Not checking if the array is full
        elementData[size++]=element;
        return true;
    }
    public E get(int index) {
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException();
        return (E) elementData[index]; // A typecast is needed here
    }

    public boolean isEmpty() {
        return size==0;
    }
    // ...
}
```

Collections and generics

- We can now create collections of different types.
- Typing is now safe. If you try to insert an object of another type, a compilation error occurs.

Class `ArrayList` with generics

```
ArrayList<Dog> cDogs = new ArrayList<>();
cDogs.insert(new Dog("Snowy"));
cDogs.insert(new Dog("Snoopy"));

ArrayList<Cat> cCats = new ArrayList<>();
cCats.insert(new Cat("Garfield"));
cCats.insert(new Cat("Heathcliff"));

cDogs.insert(new Cat("Felix")); // Compilation error!!
```

Collections and generics

- Typecasts are no longer necessary because, e.g., a collection of dogs only contains Dogs.

Class `ArrayList` with generics

```
ArrayList<Dog> cDogs = new ArrayList<>();
cDogs.insert(new Dog("Snowy"));
cDogs.insert(new Dog("Snoopy"));

Dog p = cDogs.get(1); // No typecast
System.out.println(p.getName()); // Snoopy
```

Collections and generics - Benefits

- **Safety**
  - Data types are checked at compile time.
  - This way, runtime errors are prevented.
- **Clarity**
  - Bothersome typecasts are avoided when extracting objects.
- **Understandability**
  - Data types are explicitly stated in the code, which makes the collection's purpose more obvious.

Collections and generics - The `for-each` loop

- The `for-each` loop simplifies iterating through general-purpose collections.
- It is technically called a "`for-each` loop", but the actual keyword is still `for`.
- It should be read as: "*For each* `element` *in the* `collection` *do...*"

Declaring a `for-each` loop

```
for (Type element : collection) {
    method(element);
}
```

Collections and generics - The `for-each` loop

- The two loops below work exactly the same.
- As you can see, the `for-each` syntax is more compact and clear.
- A drawback of the `for-each` loop is that you lose the `i` variable that serves as an iteration index. If you really need it, use the regular `for` instead.

`for-each` loops vs. regular loops

```
String[] arrayString = {"Hello", "World", "how", "are", "you"};

for(String s : arrayString)
    System.out.println(s);

for(int i=0; i<arrayString.length; i++)
    System.out.println(arrayString[i]);
```

Collections and generics - The `for-each` loop

- The collections in the Java API can be iterated through via iterators $\Rightarrow$ see the **Iterator Pattern** in unit 6.
- You can also iterate through them using a `for-each` loop, which is a more succinct notation for doing the exact same thing.
- Actually, any class that offers an `Iterator` can be traversed via a `for-each` loop.

`for-each` loops vs. iterator-based loops

```
String[] arrayString = {"Hello", "World", "how", "are", "you"};
List<String> listString = Arrays.asList(arrayString);

for(String s : listString)
    System.out.println(s);

for(Iterator i = listString.iterator(); i.hasNext(); )
    System.out.println(i.next());
```

Collections and generics - The `for-each` loop

- Since Java version 8, *lambda-expressions* can be used to simplify the `for-each` loop even more.
- Java collections now include a `forEach` method that takes a *lambda-expression* (i.e., a function) that is applied to every element in the collection.

`for-each` loop with *lambda-expressions*

```
String[] arrayString = {"Hello", "World", "how", "are", "you"};
List<String> listString = Arrays.asList(arrayString);

listString.forEach(n -> System.out.println(n));
```

Collections and generics

Exercise: Card game

Write the class `DeckOfCards` using `ArrayList`, generics and `for-each` loops.

Bounded types

- In generics, we only know that `T` is at least an `Object`.
- We can only call on it methods from `Object` (e.g., `toString`, `equals`, `hashCode`, `getClass`, etc.)

`Box` with generics and `Object` methods

```
public class GenericBox<T> {
    private T value;

    public void setValue (T value) { this.value = value; }
    public T getValue () { return value;}
    public GenericBox(T value) { this.value = value; }

    public static void main(String[] args) {
        GenericBox<Integer> b1 = new GenericBox<>(5);
        GenericBox<Integer> b2 = b1;
        System.out.println(b1.getClass()); // getClass is in Object
        System.out.println(b1.equals(b2)); // equals is in Object
    }
}
```

Bounded types

Bounded types

Type parameters bounded to a class that acts as an upper bound. The type parameter can only be an instance of any subclass of the bounded class, or an instance of the bounded class itself.

Bounded types

- We can know more about the `T` parameter if we restrict its possible values by declaring an upper bound using the `extends` clause thus generating a **bounded type**.
- Example: `T extends Number`. Where `T` will be any subclass of `Number` or even `Number` itself.
- Here `extends` is used in the sense of an IS_A relation, that is, it is valid for both when we inherit from a class (`extends`) and when we implement an interface (`implements`).
- As we already know that `T` is at least an instance of the superclass, we can already use on the parameter the methods of the superclass, and not only the methods defined in `Object`.

Bounded types

Bounded types example

```
public class NumericBox<T extends Number> {
    private T value;
    public void setValue(T value) { this.value = value; }
    public T getValue() { return value; }
    public NumericBox(T value) { this.value = value; }

    // doubleValue and intValue are methods of Number
    double reciprocal() { return 1 / value.doubleValue(); }
    double fraction() { return value.doubleValue() - value.intValue(); }

    public static void main(String[] args) {
        // Integer and Double are subclasses of Number
        NumericBox<Integer> n1 = new NumericBox<>(5);
        NumericBox<Double> n2 = new NumericBox<>(5.5);

        System.out.println(n1.reciprocal()); // 1/5 = 0.2
        System.out.println(n1.fraction());   // 5 => 0.0
        System.out.println(n2.reciprocal()); // 1/5.5 = 0.18181818
        System.out.println(n2.fraction());   // 5.5 => 0.5
    }
}
```

Wildcards

- We want to add a method `absEqual` to our `NumericBox`. It compares the absolute value of the current box with the absolute value inside the box passed as an argument.
- We find out that we can't compare `Integer` boxes with `Double` boxes.

Numeric box with bounded types

```
public class NumericBox<T extends Number> {
  // ...
  boolean absEqual(NumericBox<T> ob) {
    if(Math.abs(value.doubleValue()) == Math.abs(ob.value.doubleValue()))
      return true;
    else return false;
  }
  public static void main(String[] args) {
    NumericBox<Integer> n1 = new NumericBox<>(5);
    NumericBox<Double> n2 = new NumericBox<>(5.0);
    n1.absEqual(n2); // COMPILATION ERROR !!
  }                  // Asks for a NumericBox of Integer
}                    // Obtains a NumericBox of Double
```

Wildcards

Wildcards

A wildcard is a special kind of parameter that represents an unknown type when declaring generic types.

Wildcards

- **Unbounded wildcard**: `<?>`
  - Covers all classes.
- **Upper bounded wildcard:** `<? extends MyClass>`
  - Covers `MyClass` and all its subclasses.
- **Lower bounded wildcard:** `<? super MyClass>`
  - Covers `MyClass` and all its superclasses.

Wildcards

- Now the code works correctly because we ask for a numeric box with any number inside, not necessarily an `Integer`.

Numeric box with bounded types and wildcards

```
public class NumericBox<T extends Number> {
  // ...
  boolean absEqual(NumericBox<? extends Number> ob) {  // Wildcard
    if(Math.abs(value.doubleValue()) ==  Math.abs(ob.value.doubleValue()))
      return true;
    else return false;
  }
  public static void main(String[] args) {
    NumericBox<Integer> n1 = new NumericBox<>(5);
    NumericBox<Double> n2 = new NumericBox<>(5.0);
    n1.absEqual(n2); // IT WORKS !!
  }                  // Asks for an undefined NumericBox
}                    // Obtains a NumericBox of Double
```

Generics and subclasses

Valid Java code

```
String[] strArr = new String[10];
Object[] objArr = strArr;
```

What will happen here?

```
objArr[0] = new String();
objArr[1] = new Dog();
```

- We are assigning an array of Strings to an array of `Objects`, and `Object` is a superclass to `String`.
- **Options**: Compilation error, runtime error or no errors.

Generics and subclasses - Covariance

Covariance

When a subtype remains a subtype in more complex types like lists, arrays, etc.

- Arrays in Java support covariance.
- This concept did not exist in older versions of Java, as all collections consisted of elements of the same type, i.e., `Object`.
- Do generics-based collections support covariance?

Generics and subclasses - Covariance

Invalid Java code

```
List<String> strList =
        new ArrayList<String>();
List<Object> objList = strList;
```

What will happen here?

```
objList.add(new Dog());
String s = strList.get(0);
```

- Unlike arrays, a `List<String>` cannot be assigned to a `List<Object>` $\Rightarrow$ Generics-based collections are **invariant**.
- **Reason**: Otherwise, the collections would not be safe anymore (which was the whole point of generics).

Generics and subclasses - Parameters and subclasses

Universal method without generics

```
public static void printList(List l) {
    for (Object o : l) {
        System.out.println(o);  // calls o.toString()
    }
} // The base type for all collections is Object
```

Universal method using generics. Is this implementation correct?

```
public static void printList(List<Object> l) {
    for (Object o : l) {
        System.out.println(o);
    }
}
```

Generics and subclasses - Wildcards

Universal method with generics and wildcards

```
public static void printList(List<?> l) {
    for (Object e : l) { // At the very least, it will be an Object
        System.out.println(e);
    }
}
```

Using the universal method

```
List<String> strList = Arrays.asList("Hello","World","how","are","you");
List<Integer> intList = Arrays.asList(5, 3, 9, 8);

printList(strList);
printList(intList);
```

Generics and subclasses - Wildcards

- On the previous slide, we only know that an element is, at the very least, an `Object`.
- We have limited freedom over what we can do with it: passing it as a parameter, returning it, calling `Object` methods such as `equals`, `hashCode`, `toString`, etc.
- For example, if we pass a list of `Strings`, `printListn` cannot take advantage of `String` methods such as `charAt()`, `toUpperCase()`, etc.
- This is why the `extends` and `super` wildcards are necessary.

Generics and subclasses - Wildcards

- Recall the class `Figure`, with its methods `area()` and `perimeter()` and the subclasses `Circle` and `Rectangle`
- We want to write a method that takes a list of figures and iterates through them, adding their areas.

Is this implementation correct?

```
public static double addAreas(List<Figure> l) {
    double sum = 0;
    for (Figure f : l)
        sum += f.area();
    return sum;
}
```

- It is **correct** in that it meets its goal, but it is **limited**. It only accepts a list of instances of `Figure`, and not subclasses like `Circle` or `Rectangle`.

Generics and subclasses - Wildcards

Limitations of `addAreas`

```
Circle c1 = new Circle(5);
Circle c2 = new Circle(3);
Rectangle r1 = new Rectangle(5,6);

List<Figure> figList = Arrays.asList(c1, c2, r1);
List<Circle> cirList = Arrays.asList(c1, c2);

System.out.println("The sum is = " + sumAreas(figList));
System.out.println("The sum is = " + sumAreas(cirList));
```

- **Compilation error** on the last line of code. We cannot pass a `List<Circle>` where a `List<Figure>` is demanded.

Generics and subclasses - Wildcards

- The `extends` wildcard is what we need here, i.e., creating a method that can add the areas of a list of figures or any subclass of figure.
- `extends` is a *bounded wildcard* that is linked to a class.

`extends` wildcard

```
public static double sumAreas(List<? extends Figure> l) {
    double sum = 0;
    for (Figure f : l)
        sum += f.area();  // => See dynamic binding
    return sum;
}
//...
    List<Figure> figList = Arrays.asList(c1, c2, r1);
    List<Circle> cirList = Arrays.asList(c1, c2);

    System.out.println("The sum is = " + sumAreas(figList));
    System.out.println("The sum is = " + sumAreas(cirList));
```

Generics and subclasses - Wildcards

- Wildcards have restrictions too.
- Example: We want to write a method that takes a list of figures or any of its subclasses and then inserts a `Rectangle` into it.

Is this code valid?

```
public static void insertRectangle(List<? extends Figure> l) {
    l.add(new Rectangle());
}
```

- **NO**. We cannot add a `Rectangle` to a list if we are not completely sure of the type of its elements.
- For example, we would be corrupting a list of `Circles`, and our collections were supposed to be safe.

Generics and subclasses - Wildcards

Conclusion

The `<?>` and the `<? extends MyClass>` wildcards treat collections as read-only.

- For the former wildcard, we know that it represents a collection of `Object` or any of its subclasses.
- For the latter wildcard, we know that it represents a collection of `MyClass` or any of its subclasses.
- Inserting an element into any of these collections runs the risk of corrupting the collection.

Generics and subclasses - Wildcards

- What if we wrote a method that takes a list of figures or any of its **superclasses** and then inserts a `Rectangle` into it?

Would this code be valid?

```
public static void insertRectangle(List<? super Figure> l) {
    l.add(new Rectangle());
}
```

- **YES**. There is no impediment whatsoever to inserting the `Rectangle`.

Generics and subclasses - The *Get* and *Put* Principle

Get and *Put* Principle (aka *Producer-Consumer*)

- Use the `extends` wildcard when you just want to read values from a collection.
- Use the `super` wildcard when you just want to write values to a collection.
- Do not use any wildcards when you want to both read values from a collection and write values to it.

Generic methods

- Suppose we want to write a static function that copies the contents of a list to another list.
- The non-generic version runs smoothly because both are lists of `Object`.

Version without generics (class `Collections`)

```
public static void copy(List dest, List src)
```

Generic methods

Version with unbouded wilcards. Is it valid?

```
public static void copy(List<?> destination, List<?> origin)
```

- This would not work. The destination list uses the <?> wildcard, which makes it read-only.
- We need to be able to indicate that the type in the origin and the destination is the same, but it can be any.

Generic methods

Generic methods

Methods that, analogous to classes, are parameterized by a type parameter located before the return type.

Version with generic methods

```
public static <T> void copy(List<T> dest, List<T> src)
```

- The type parameter "presents" to the method a parameter that will be used in its declaration.
- In the example we indicate to the method that `T` is a generic parameter and that it has to be the type of both lists.
- This solution is correct but also limited. For example, it can only copy a list of `Dogs` to another list of `Dogs`.

Generic methods

Version with generic methods and bounded wildcards

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

- We can apply the **Get and Put Principle** to make the method more flexible.
- The `src` list is read-only, so we will use the `extends` wildcard.
- The `dest` list is write-only, so we will use the `super` wildcard.
- Now we can copy a list of `GermanShepherd`s into a list of `Animal`s (i.e., a subclass of `Dog` into a superclass of `Dog`).

Generic methods

Version with generic methods and bounded wildcards

```
class Animal { /* ... */ }
class Dog extends Animal { /* ... */ }
class GermanShepherd extends Dog { /* ... */ }

public class Copy {
    public static void main(String[] args) {
        ArrayList<Animal> animals = // ...
        ArrayList<Dog> dogs = // ...
        ArrayList<GermanShepherd> germanShepherds = // ...

        Collections.copy(animals, germanShepherds); // IT WORKS !!
    }
}
```

Interfaces and generics

- It is also possible to define generic interfaces like we did with classes, including a type parameter that can then be used in its methods.
- Many interfaces in the Java API now support generics. For example, `Iterator`.

Version without generics

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Version with generics

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Interfaces and generics

- We can implement an interface in its generic form if the class that implements it is also generic.

Generic classes and generic interfaces

```
class MyGenericIterator<E> implements Iterator<E> {
    public boolean hasNext() { /* ... */ }
    public E next() { /* ... */ }
    public void remove() { /* ... */ }
}
```

Interfaces and generics

- If the class that implements the generic interface is not generic, you can implement the interface, but using a specific type and not a generic type.

Non-generic classes and generic interfaces

```
class MyIntegerIterator implements Iterator<Integer> {
    public boolean hasNext() { /* ... */ }
    public Integer next() { /* ... */ }
    public void remove() { /* ... */ }
}
```

Interfaces and generics

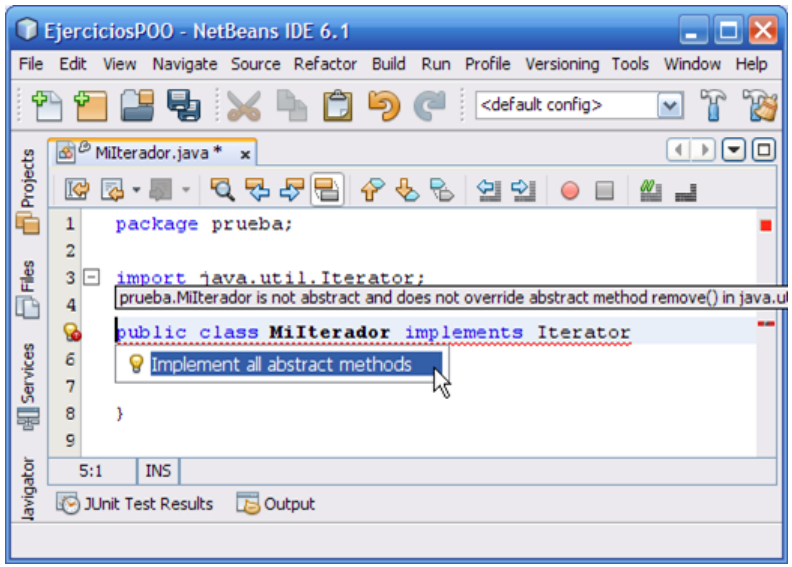
Method `sort` of class `Collections`

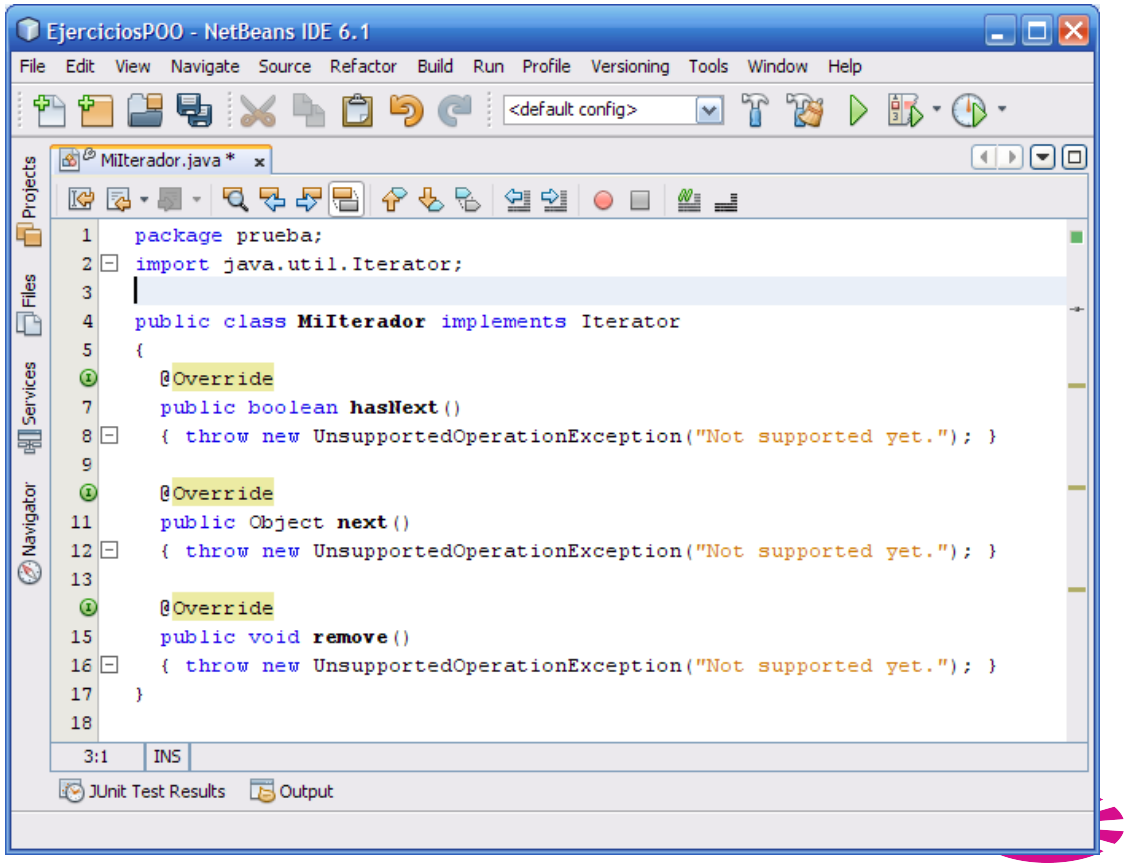
```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

- `sort` allows you to sort elements `T` as long as these elements implement the interface `Comparable` on elements of `T` or one of its superclasses.
- e.g. I can sort a list of `Dog`s if this class implements `Comparable` on `Dog` or on a superclass like `Animal`.

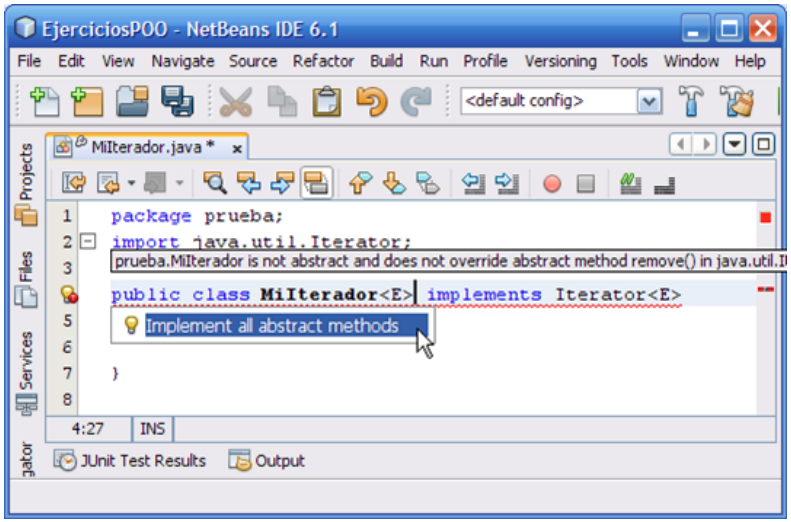
Interfaces and generics

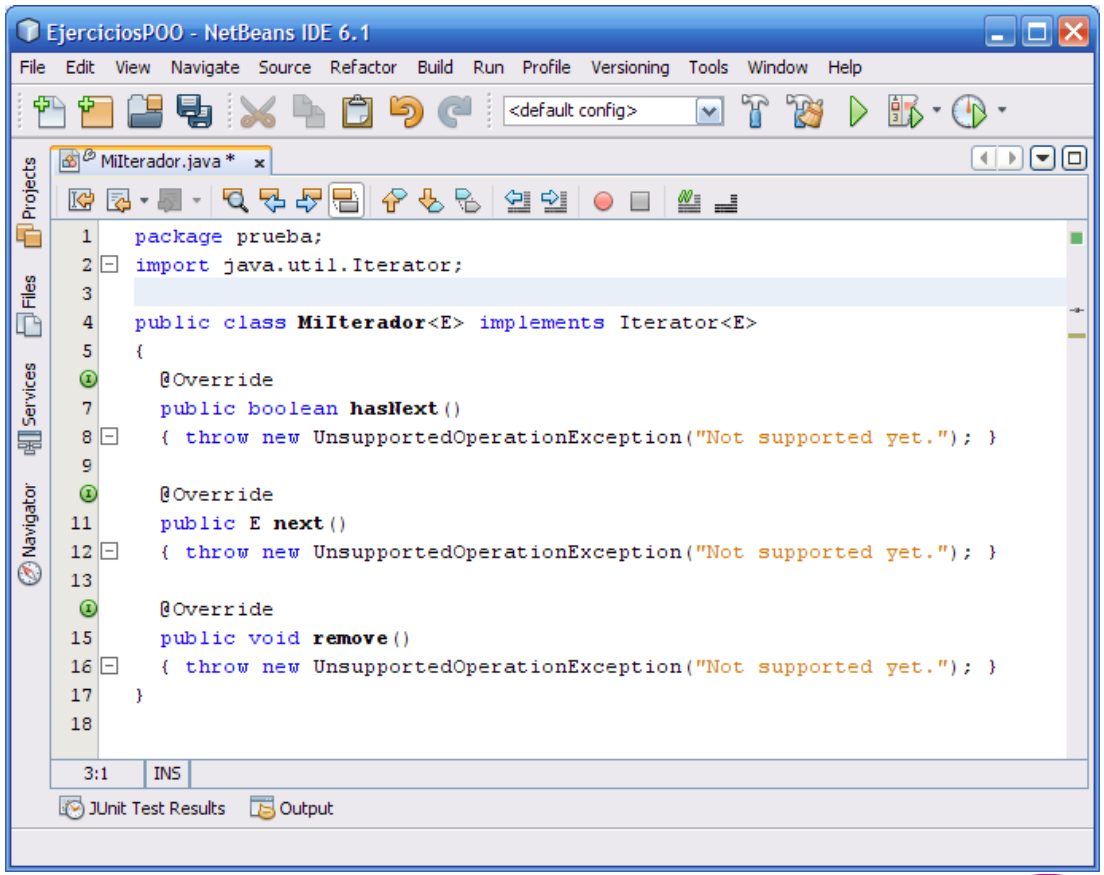
- Careful with the IDE's autocomplete feature.





Interfaces and generics





Complexities of generics

1. **Code that was simple (and unsafe) is now more complex (but safe).**

Example

```
// Java 4: Throws exceptions if the objects in the list are not comparable
public static void sort(List list)

// Java 5 onwards: Compilation fails if the objects are not comparable.
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

2. **How do we maintain backwards compatibility?**
   - Code written for collections without generics must still work now that the collections use generics.

Complexities of generics - Backwards compatibility

1. **We break backwards compatibility**.
   - The code written for version 4 of Java would not be valid for version 5 unless it is updated.
   - Something similar to what happened with Python between versions 2 and 3. This is undesirable for a language as widespread as Java.
2. **We have two versions of the API coexisting**.
   - One with generics and one without generics (similar to C#).
   - The old code can use the old API and the new one the new API, but it duplicates the API and complicates the interoperability between both: version problems, the new code doesn't work on old platforms, etc.
3. **Generics only really exist at compile time**.
   - It is the **solution adopted by Java**.
   - Known as *type erasure*, all information about generics are not actually propagated to the bytecodes.

Complexities of generics - Backwards compatibility

- Generics only really exist at compile time but not at runtime.

Erasure process

```
System.out.println("runtime type of ArrayList<String>: " +
                                   new ArrayList<String>().getClass());
System.out.println("runtime type of ArrayList<Long>  : " +
                                   new ArrayList<Long>().getClass());

//prints: runtime type of ArrayList<String> : class java.util. ArrayList
//        runtime type of ArrayList<Long>   : class java.util. ArrayList
```

Complexities of generics - Backwards compatibility

- **Rationale of *erasure***
  - **Compatibility**: there are no differences between bytecode with generics and the same bytecode without generics. Old code works on new JVMs and compiled new code works on old JVMs.
  - **Evolution rather than revolution**: we can apply generics to an API (e.g., Collections) without updating the clients that use it.
- **Drawbacks of *erasure***
  - You can use – perhaps unintentionally – an API with and without generics at the same time $\Rightarrow$ confusing; raises the `unchecked` warning.
  - Minor problems arising from the absence of runtime generics:
    - It is impossible to effectively overload methods that use generics. `method(List<Integer> x)` is the same as `method(List<String> x)` at runtime.
    - We may even breach type safety using RTTI (Run Time Type Information) methods maliciously.

Complexities of generics - The `unchecked` *warning*

- This *warning* occurs when you use a generics-based collection but you are not using generics properly.
- Compilation succeeds due to backwards compatibility, but a *warning* is issued to remind us that we should use generics.
- If we are using collections with generics the warning should never be issued.

Examples

```
List<Animal> aniList = new ArrayList();  // Unchecked warning
List aniList2 = new ArrayList<Animal>(); // No warning at this point
aniList2.add(new Cat("Garfield"));       // It is issued here, though
```

Complexities of generics - The `unchecked` *warning*

- If you compile using the option `-Xlint:unchecked` you can see detailed information about the *warning*.

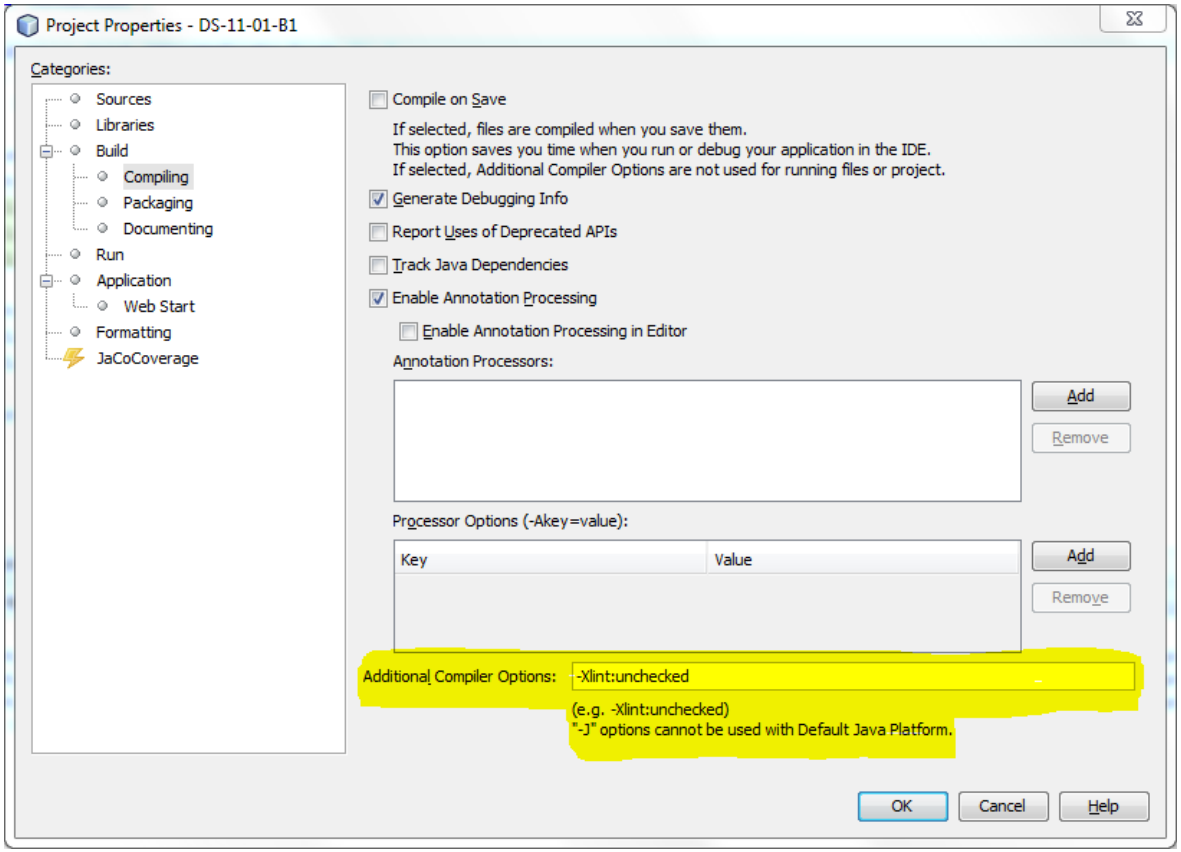


Table of Contents

1 Abstraction and encapsulation

2 Modularity

3 Hierarchy

4 Polymorphism

5 Typing
- Ways of Typing
- Restrictions of Java Typing
- Duck typing

6 Dynamic binding

Typing

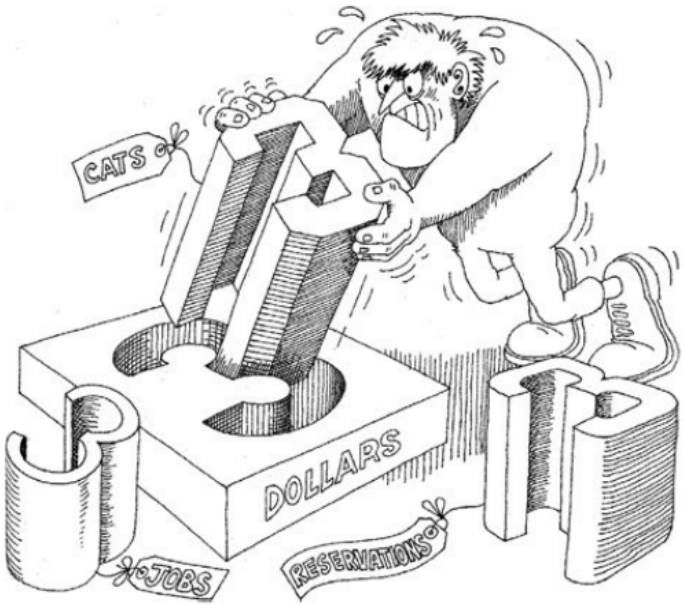
Types

A type is a precise characterization of the structural and behavioral properties that are shared by a number of entities.

- In OOPLs, a type is usually a class, but not always:
  - There are types that are not classes (primitive types like `int`, `float`, etc.).
  - Sometimes, different classes implement the same data type (e.g., the type `List` is implemented by `ArrayList` and `LinkedList`).
  - And sometimes, a subclass might not even be a subtype ⇒ See **U5 - SOLID principles: Liskov's Substitution principle**.

Typing

- **Main goal: Concept of congruence**
  - For example, string concatenation returns a string, counting the number of characters in a string returns an integer, etc.
  - This helps to detect errors and facilitates the internal functioning of compilers.
- **Dimensions**
  - Static and Dynamic.
  - Strong and Weak.



Strong typing prevents mixing of abstractions.

Static and dynamic typing

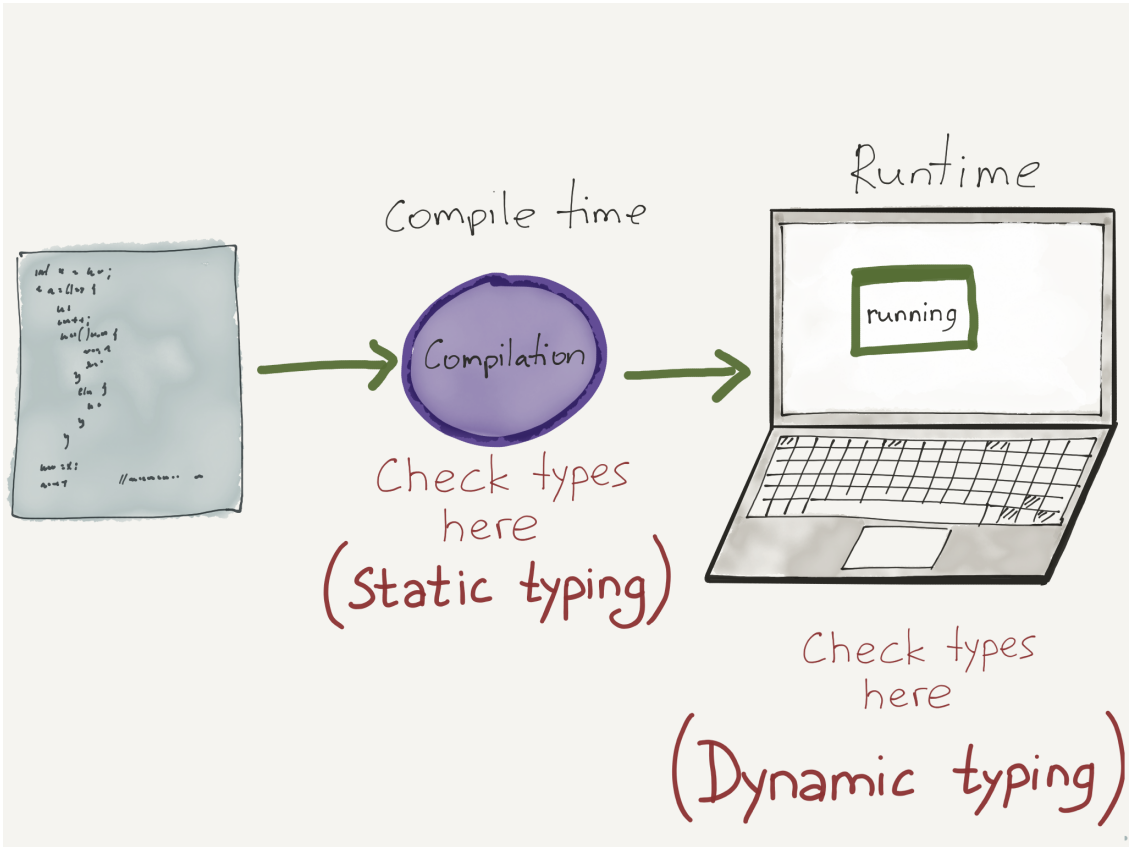
- It refers to the moment at which type checking is carried out.

Static typing

Types are checked at compile time.

Dynamic typing

Types are checked at runtime.



Static and dynamic typing

- **Static typing**
  - **Advantages**: Better performance, better code readability, errors are detected earlier and more easily.
  - **Disadvantages**: The code is more rigid and verbose.
  - **Examples**: C, Java, C#, C++.
- **Dynamic typing**
  - **Advantages**: More succinct code, easier to write and learn.
  - **Disadvantages**: Errors take longer to be detected and are more complex to fix.
  - **Examples**: Python, JavaScript, PHP, Ruby.

Java

```
String s = "Hello, World";
```

Python

```
s = "Hello, World"
```

Strong and weak typing

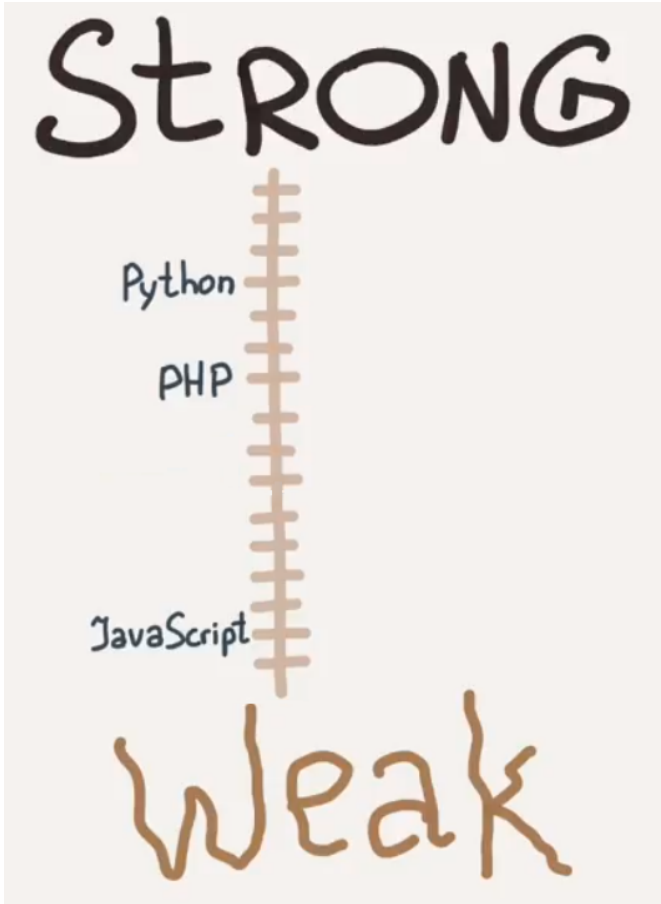
- It refers to how permissive the language is when analyzing type expressions and making implicit conversions between them.
- It is not a binary value but a scale.

Strong typing

Typing rules are strict.

Weak typing

Typing rules are more flexible.



Strong and weak typing

- **Strong typing**
  - **Advantages**: Fewer mistakes are made.
  - **Disadvantages**: Certain operations may not be carried out as they are considered erroneous.
  - **Examples**: Java, Python.
- **Weak typing**
  - **Advantages**: The code is easier to write.
  - **Disadvantages**: More fragile code and more unpredictable results.
  - **Examples**: C, JavaScript.

Java

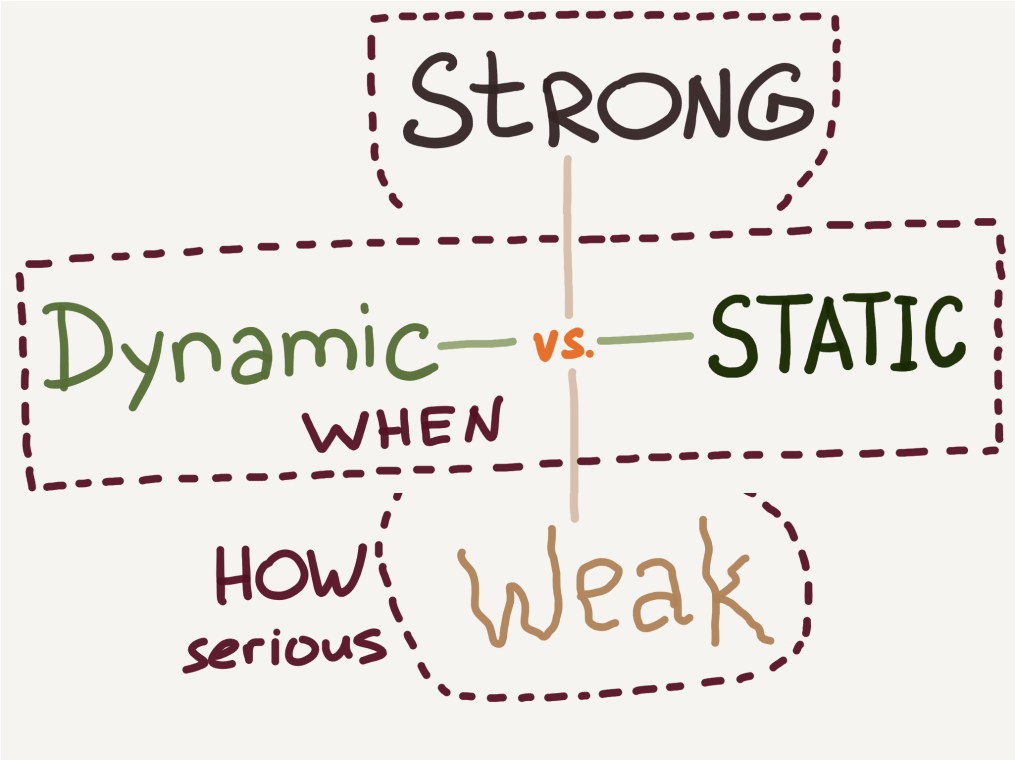
```
int a = 4 + '7'; // 59 (7 is 55 in ASCII)
int b = 4 * "7"; // compilation error
int c = 2 + true;  // compilation error
int d = false - 3; // compilation error
```

JavaScript

```
4 + '7';      // '47'
4 * '7';      // 28
2 + true;     // 3
false - 3;    // -3
```

Dimensions of typing

- The two dimensions are independent.



- **Static and strong**: Java.
- **Static and weak**: C.
- **Dynamic and strong**: Python.
- **Dynamic and weak**: JavaScript.

Restrictions of Java Typing

(1) Declaration restriction

All the entities in the PL (variables, objects, attributes, parameters) must have a declared type.

- Sometimes it is not necessary to declare the type because it is inferred by the compiler.

Inferred types

```
// The parametric type of the ArrayList is inferred from the List
List<String> listString = new ArrayList<>();

// Lambda-expression from Java 8. Even though the type of "n" is never
// stated, it is inferred from the type of the list (i.e., String)
listString.forEach(n -> System.out.println(n));

// Local variable type inference (since Java 10)
var isValid = true;
if (isValid) System.out.println("valid");
```

Restrictions of Java Typing

(2) Compatibility restriction

On any `x = y` assignment, and on any call that passes `y` as an argument for the formal parameter `x`, the type of `y` must be compatible with `x`.

- The concept of compatibility in OOPLs is mainly based on inheritance and inclusion polymorphism: `y` is compatible with `x` if it is a descendant of `x`.

Compatibility through inheritance

```
// Dog is compatible with Animal because it is one of its subclasses
Animal a = new Dog();
```

Restrictions of Java Typing

(3) Feature call restriction

In order to use a feature (i.e., a method or an attribute) from a class X, this feature must be defined for X or any of its ancestors.

What is the result of running this code?

```
class Animal { /* The method "bark" has not been defined for Animal */ }

class Dog extends Animal {
    public void bark() { System.out.println("Woof"); }

    public static void main(String[] args) {
        Animal anAnimal = new Dog();
        anAnimal.bark();
    }
}
```

Duck Typing

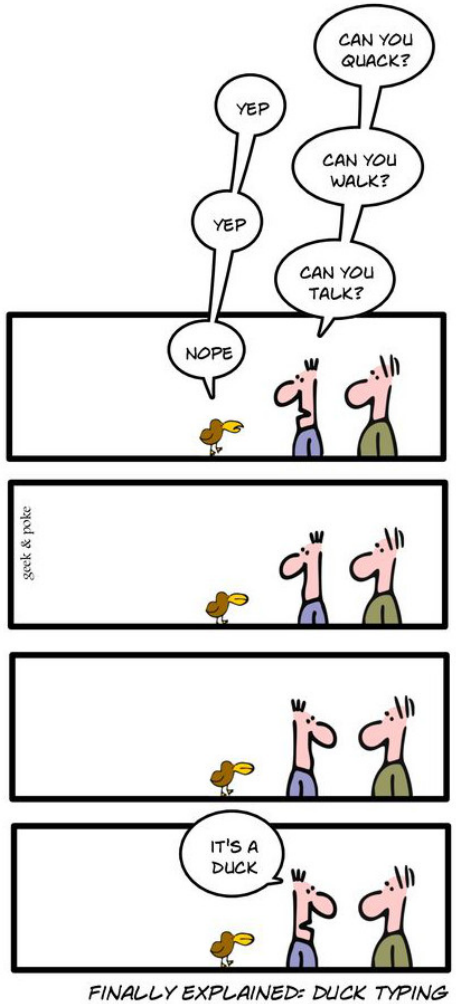
Duck Test
(James Whitcomb Riley, poet)

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Duck Typing

Duck Typing

Type checks consist in checking that the object at hand has the desired characteristics (*walks*, *swims*, *quacks*), rather than checking the type of the object.



Duck Typing

- *Duck typing* is characteristic of dynamically-typed PLs.

Duck typing in *Python*

```
class Duck:
    def quack(self):
        print "Quack, quack!"
    def fly(self):
        print "Flap, Flap!"

class Person:
    def quack(self):
        print "I'm Quackin'!"
    def fly(self):
        print "I'm Flyin'!"

def in_the_forest(mallard):  // mallard does not have a declared type
mallard.quack()
mallard.fly()

in_the_forest(Duck())    // Prints "Quack, quack! Flap, Flap!"
in_the_forest(Person()) // Prints "I'm Quackin'! I'm Flyin'!"
```

Duck Typing in Java

- Java's static typing means that it cannot support *duck typing*.

Duck typing in *Java*

```
class Duck {
  public void quack() {
    System.out.println("Quack, quack!");
  }
  public void fly() {
    System.out.println("Flap, Flap!");
  }
}

class Person {
  public void quack() {
    System.out.println("I'm Quackin'!");
  }
  public void fly() {
    System.out.println("I'm Flyin'!");
  }
}
```

Violates restriction no. 2

```
class DuckTyping {
  // function declares a Duck type
  public static void in_the_forest
                          (Duck mallard) {
    mallard.quack();
    mallard.fly();
  }

  public static void main(String[] args){
    in_the_forest(new Duck());
    in_the_forest(new Person()); // ERROR
    // Person is not compatible with Duck
}
```

Duck Typing in Java

- Java's strict typing means that it cannot support *duck typing*.

Duck typing in *Java*

```
class Duck {
  public void quack() {
    System.out.println("Quack, quack!");
  }
  public void fly() {
    System.out.println("Flap, Flap!");
  }
}

class Person {
  public void quack() {
    System.out.println("I'm Quackin'!");
  }
  public void fly() {
    System.out.println("I'm Flyin'!");
  }
}
```

Violates restriction no. 3

```
class DuckTyping {
  // function declares and Object type
  public static void in_the_forest
                        (Object mallard) {
    mallard.quack(); // ERROR
    mallard.fly();   // ERROR
    // quack() and fly() are not defined
    // for Object
  }

  public static void main(String[] args){
    in_the_forest(new Duck());
    in_the_forest(new Person());
}
```

Why Java cannot support *duck typing*

- Even though some type manipulations may be valid in some contexts, it is preferable to forbid them unless you are completely sure.
- The point is that it is preferable to detect errors at compile time.
- **Example:**
  - Both `Duck` and `Person` have the methods `quack()` and `fly()`, so they could theoretically be passed as an argument for the function `in_the_forest(Object mallard)` without any problems.
  - However, that function accepts any kind of object, and not all objects have the methods `quack()` and `fly()`. Therefore, the function could fail at runtime in those cases.

How to solve the *duck typing* problem in Java

(1) Using explicit type casts ⇒ runtime error

```
class Duck {
  public void quack() {
    System.out.println("Quack, quack!");
  }
  public void fly() {
    System.out.println("Flap, Flap!");
  }
}

class Person {
  public void quack() {
    System.out.println("I'm Quackin'!");
  }
  public void fly() {
    System.out.println("I'm Flyin'!");
  }
}
```

```
class DuckTyping {
  public static void in_the_forest
                          (Object mallard) {
    ((Duck)mallard).quack();
    ((Duck)mallard).fly();
  }

  public static void main(String[] args){
    in_the_forest(new Duck());   // OK
    in_the_forest(new Person()); // ERROR
}
```

Can cause a runtime error

How to solve the *duck typing* problem in Java

(2) Using `instanceof` ⇒ antipattern

```
class Duck {
  public void quack() {
    System.out.println("Quack, quack!");
  }
  public void fly() {
    System.out.println("Flap, Flap!");
  }
}

class Person {
  public void quack() {
    System.out.println("I'm Quackin'!");
  }
  public void fly() {
    System.out.println("I'm Flyin'!");
  }
}
```

```
class DuckTyping {
  public static void in_the_forest
                          (Object mallard) {
    if (mallard instanceof Duck) {
      ((Duck)mallard).quack(); // OK
      ((Duck)mallard).fly();   // OK
    } else if(mallard instanceof Person){
      ((Person)mallard).quack(); // OK
      ((Person)mallard).fly();   // OK
    } else {
      // ...
    }
  }
}
```

NO!! Inquiring about an object's type and then doing different things based on that is an **OO anti-pattern**

How to solve the *duck typing* problem in Java

(3) Using interfaces ⇒ a more OO solution

```
interface QuackAndFly {
    public void quack();
    public void fly();
}
class Duck implements QuackAndFly {
  public void quack() {
    System.out.println("Quack, quack!");
  }
  public void fly() {
    System.out.println("Flap, Flap!");
  }
}
class Person implements QuackAndFly {
  public void quack() {
    System.out.println("I'm Quackin'!");
  }
  public void fly() {
    System.out.println("I'm Flyin'!");
  }
}
```

```
class DuckTyping {
  public static void in_the_forest
                    (QuackAndFly mallard) {
    mallard.quack();
    mallard.fly();
  }

  public static void main(String[] args){
    in_the_forest(new Duck());   // OK
    in_the_forest(new Person()); // OK
  }
}
```

A more OO solution. Works thanks to **dynamic binding**

Table of Contents

1 Abstraction and encapsulation

2 Modularity

3 Hierarchy

4 Polymorphism

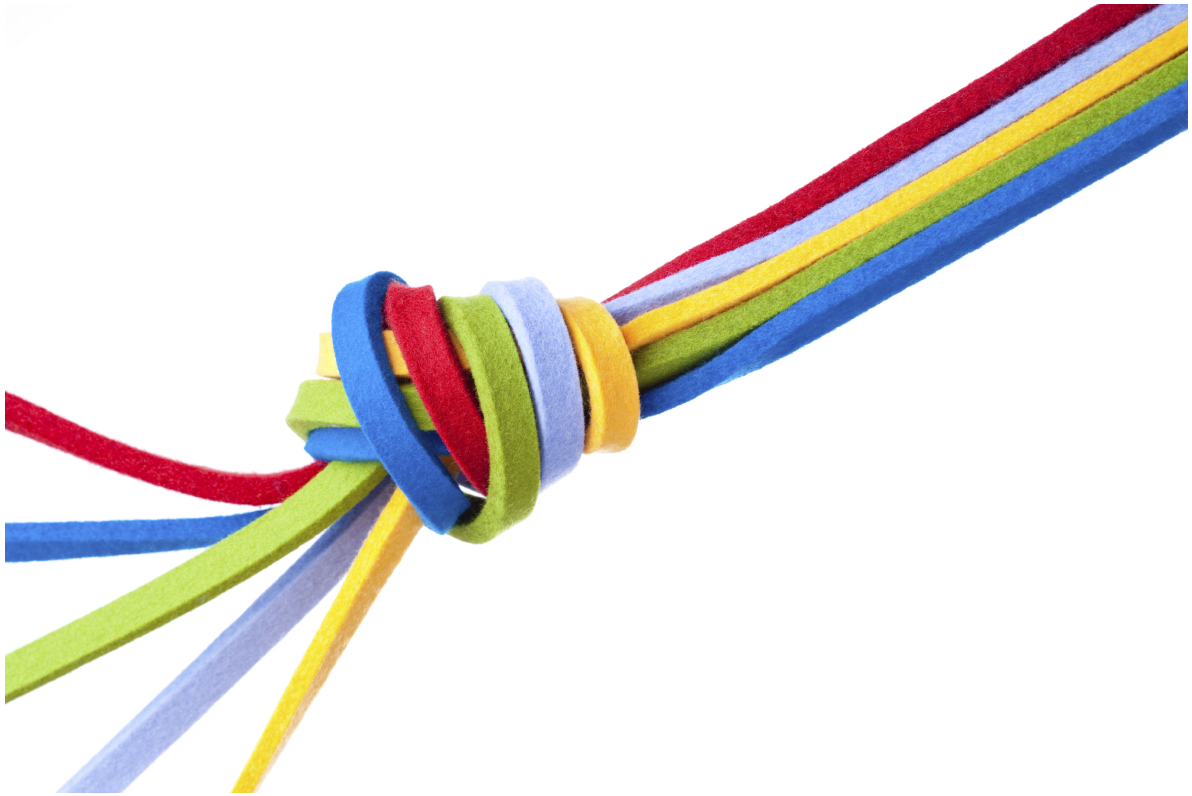
5 Typing

6 Dynamic binding
- Functioning
- The Essence of OO Design
- Methods without Dynamic Binding

Dynamic binding

Binding

The process that links a call to a method (or message) with the code that is ultimate executed.



Types of binding

Static (or early) binding

- Binding is done at compile time, based on the declared type of the object.
- Used in Java by static methods and instance methods that are private and final (as the latter cannot be overridden).

Dynamic (or late) binding

- Binding is done at runtime, based on the dynamic type of the object at runtime. This is what determines which version of the method will be run.
- Used in Java by instance methods that are neither private nor final.

Example of dynamic binding

- Let us recall the abstract class `Figure` with two abstract methods: `area` and `perimeter`. Consider also the subclasses `Circle` and `Rectangle`, which implement said methods.

Class `Figure`

```
public abstract class Figure {
    public abstract double area();
    public abstract double perimeter();
}
public class Circle extends Figure {
    public double area() { return Math.PI*radius*radius;  }
    public double perimeter() { return 2*Math.PI*radius; }
    // ...
}
class Rectangle extends Figure {
    public double area() { return base*height; }
    public double perimeter() { return (base*2)+(height*2); }
    // ...
}
```

Example of dynamic binding

- Recall also the method `addAreas`, which added the respective areas of a list of figures.

Method `addAreas`

```
public static double addAreas(List<? extends Figure> l) {
    double sum = 0;
    for (Figure f : l) {
        sum += f.area();
    }
    return sum;
}
```

Calling `area` executes different methods depending on the dynamic type of each figure

- We are certain that the `area` method will be present, as that method is defined for the superclass `Figure` (albeit abstractly).

Example of dynamic binding

- Suppose we now create a new subclass of `Figure` called `Square`.

Class `Square`

```
class Square extends Figure {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    public double area() {
        return side * side;
    }

    public double perimeter() {
        return side * 4;
    }
    // ...
}
```

Example of dynamic binding

- We can now check that the `addAreas` method works correctly with the class `Square` and calls its `area` method.

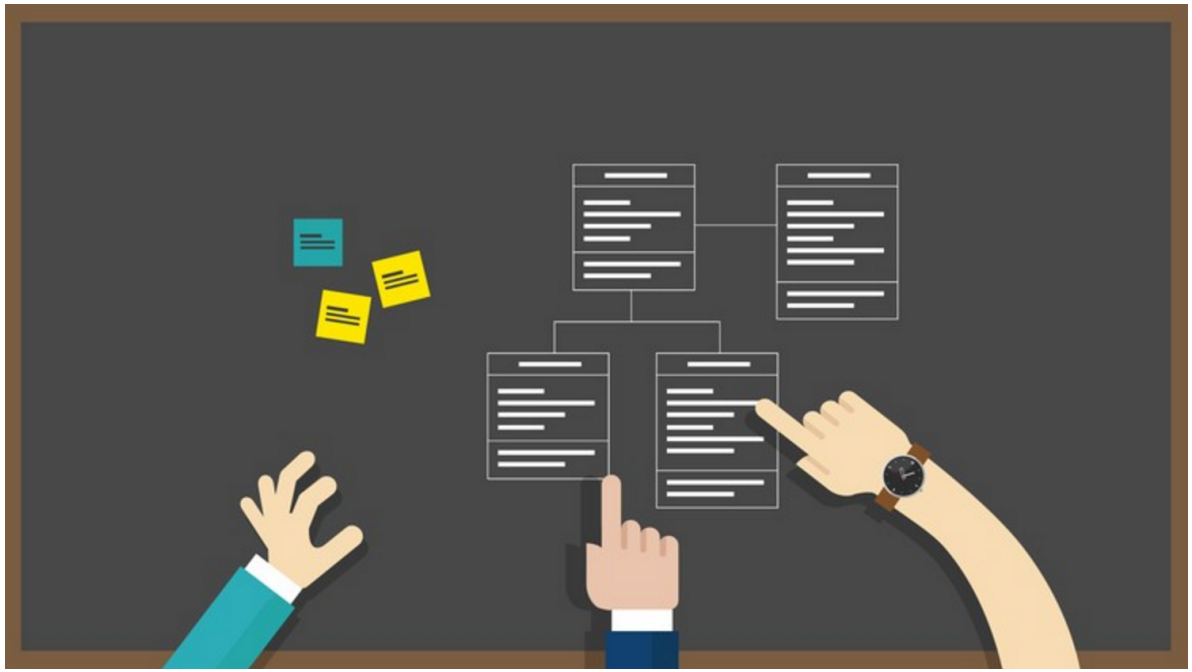
Class `Square`

```
List<Figure> l = new ArrayList<>();
l.add(new Rectangle(2, 3));
l.add(new Circle(3));
l.add(new Rectangle(5, 4));
l.add(new Square(2));

System.out.println("Sum = " + sumAreas(l));
```

The essence of OO design

The essence of OO design

Use the typical properties of OO (**inheritance**, **polymorphism** and **dynamic binding**) jointly to obtain the advantages that OO offers: **flexibility**, **scalability**, **extensibility**, **abstraction**, etc.



The essence of OO design

IMPORTANT !!
Almost all the examples of design principles and patterns in units 5 and 6 are variations of this example.

- **Properties**:
  - **Inheritance**: used to describe the relations between figures.
  - **Polymorphism**: used to create a diverse list of figure types.
  - **Dynamic binding**: used to call the `area` method.
- **Advantages**:
  - We are writing code (i.e., `sumAreas`) that will be able to work with future classes (e.g., `Square`) that haven't been created yet ⇒ **Flexibility**, **Scalability**, **Extensibility**, etc.
  - The `addAreas` method is oblivious to the specific classes it is using; all it knows is that they are figures ⇒ **Abstraction**, **Safety**, **Maintainability**, etc.

Example of dynamic binding

What is the result of this code?

```
abstract class Animal {
  public void makeNoise() { System.out.println("I make no noise"); }
}
class Dog extends Animal {
  public void makeNoise() { bark(); }
  public void bark() { System.out.println("Woof"); }
}
class Fish extends Animal { /* Does not redefine makeNoise */ }
    //...
    Animal a1 = Dog();
    Animal a2 = Fish();
    a1.makeNoise();
    a2.makeNoise();
    //...
}
```

Dynamic binding and static methods

What is the result of this code?

```
class Parent {
    public static void staticMethod()
        { System.out.println("Parent");}
}
class Child extends Parent {
    public static void staticMethod()
        { System.out.println("Child"); }
}
class Statics {
    public static void main(String[] args) {
        Parent p = new Child();
        p.staticMethod();
    }
}
```

Dynamic binding and static methods

- Static methods use static binding, so they do not take advantage of OO.
- Nevertheless, a static method can be useful in certain contexts:
  - When the method makes no use of attributes.
  - When we want to call it without creating an instance.
  - When the result only depends on the arguments (e.g., `public int factorial(int number)`).
  - Some design patterns, such as **Factory Method**.

Dynamic binding and final methods

What is the result of this code?

```
class Parent {
    public final void finalMethod() {
        System.out.println("Parent");
    }
}
class Child extends Parent {
    public final void finalMethod() {
        System.out.println("Child");
    }
}
 class Finals {
    public static void main(String[] args) {
        Parent p = new Child();
        p.finalMethod();
    }
}
```

Dynamic binding and final methods

Final classes

Classes that cannot be extended through inheritance.

Final methods

Methods that cannot be overridden by subclasses.

- A final method can belong to a non-final class.
- In fact, methods in final classes are implicitly final, since it is impossible to override them.

Dynamic binding and final methods

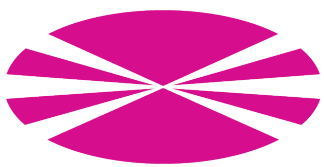
- **Final classes**
  - Final classes limit the extensibility of a class and should be avoided unless your intention is to limit extensibility $\Rightarrow$ **immutable classes** must be final to ensure no subclass breaks the immutability.
- **Final methods**
  - Final methods also place limitations on extending (e.g., overriding) the behavior of a class.
  - Sometimes we don't need to prohibit a class extension entirely, but only prevent overriding of some methods $\Rightarrow$ **Template-method pattern** (Unit 6).

Unit 3: Basic Properties of Object Orientation

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science, Faculty of Computer Science



UNIVERSIDADE DA CORUÑA