OOP Course

ANDRES MASEGOSA

Today's Plan

1. Introduction to the Course
2. Introduction to Java

Lecturer

- **Research in Trustworthy Machine Learning:**
  - Probabilistic Approach.
  - Robust Machine Learning methods.
  - Probabilistic Programming Languages
  - Theoretical Machine Learning.

- **Software Development Experience:**
  - Amidst Toolbox (www.amidsttoolbox.com) in Java.
  - InferPy (https://inferpy.readthedocs.io/) in Python.

https://andresmasegosa.github.io/

Teaching Assistant

Stefan Profft Larsen

spla21@student.aau.dk

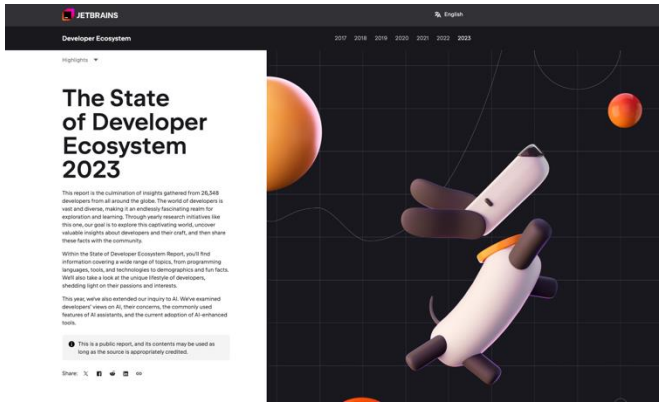
Course Overview

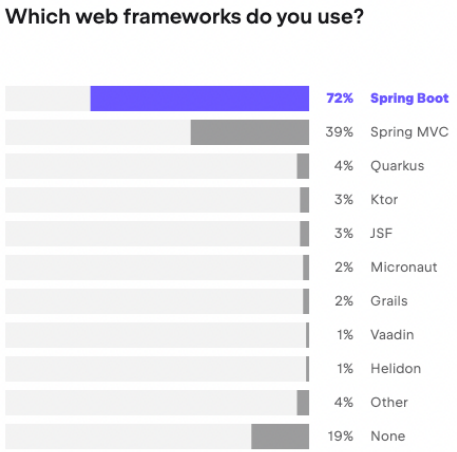
- **5 ECTS/11 Sessions**
- **Objective:**
  - Learn the essential concepts and structuring mechanisms within OOP languages
- **Knowledge:**
  - Knowledge of concepts within the object-oriented programming paradigm
- **Skills:**
  - construct OOP programs using OOP concepts.
  - reason about program design and design patterns
  - systematic testing
- **Competencies:**
  - design, document and test a large object-oriented program
  - analyze and discuss a program based on object-oriented design principles and concepts
  - define and discuss key concepts in object-oriented programming

9 Lecture – Exercise Sessions:

- Lecturing + Live Coding + Small Exercises
- Group Exercises.

1 Workshop WebPage – Spring Boot





- You will be introduced to Java Spring Boot Framework
- 24th October (the day before the Hackathon Week).
- Stephan (TA) will lead the workshop

Exam

- **Written Exam:**
  - Several programming exercises
    - Similar to the ones made in class
  - Books, notes, etc. will be allowed.
  - **Generative AI tools not!!**
  - January,
  - Internal censor,
  - Graded by the 7-point scale

1 Session Exam Rehearsal

- You will be presented several exercises similar to the exam.
- We will do the exercises, and they will be corrected in class.

Tentative Course Plan

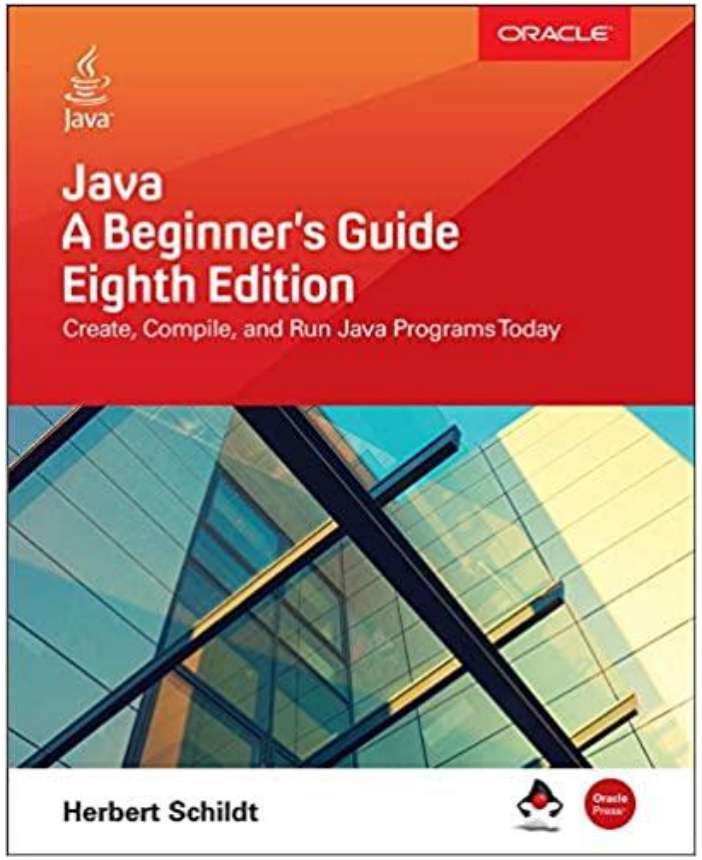
Session	Lecture topic	Type	Date	Hours	Room
1	*Intro to Java + IntelliJ + Debbuger*	Lecture+Exercises	4/9	4	0.091
2	Classes Objects + Imports	Lecture+Exercises	11/9	4	0.091
3	Encapsultaiton + Inheritance	Lecture+Exercises	25/9	4	0.091
4	Object Oriented Design	Lecture+Exercises	2/10	4	0.091
5	Generics and Java Collections	Lecture+Exercises	9/10	4	0.091
7	Unit Test - TDD	Lecture+Exercises	23/10	4	0.091
6	Workshop WebPage	Workshop	24/10	4	0.091
8	Exceptions and Enums	Lecture+Exercises	6/11	4	0.091
9	Design Patterns I	Lecture+Exercises	13/11	4	0.091
10	Design Patterns II	Lecture+Exercises	20/11	4	0.091
11	Exam Preparation	Workshop	27/11	4	0.091

Literature

Java: A Beginner's Guide, Eighth Edition

de Herbert Schildt (Author)

★★★★½ 255 calificaciones

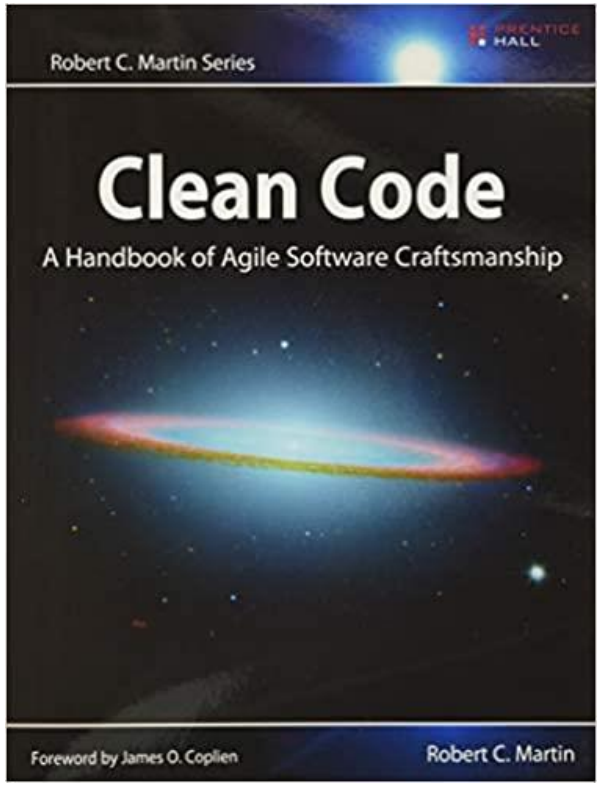


[Optional] Literature II

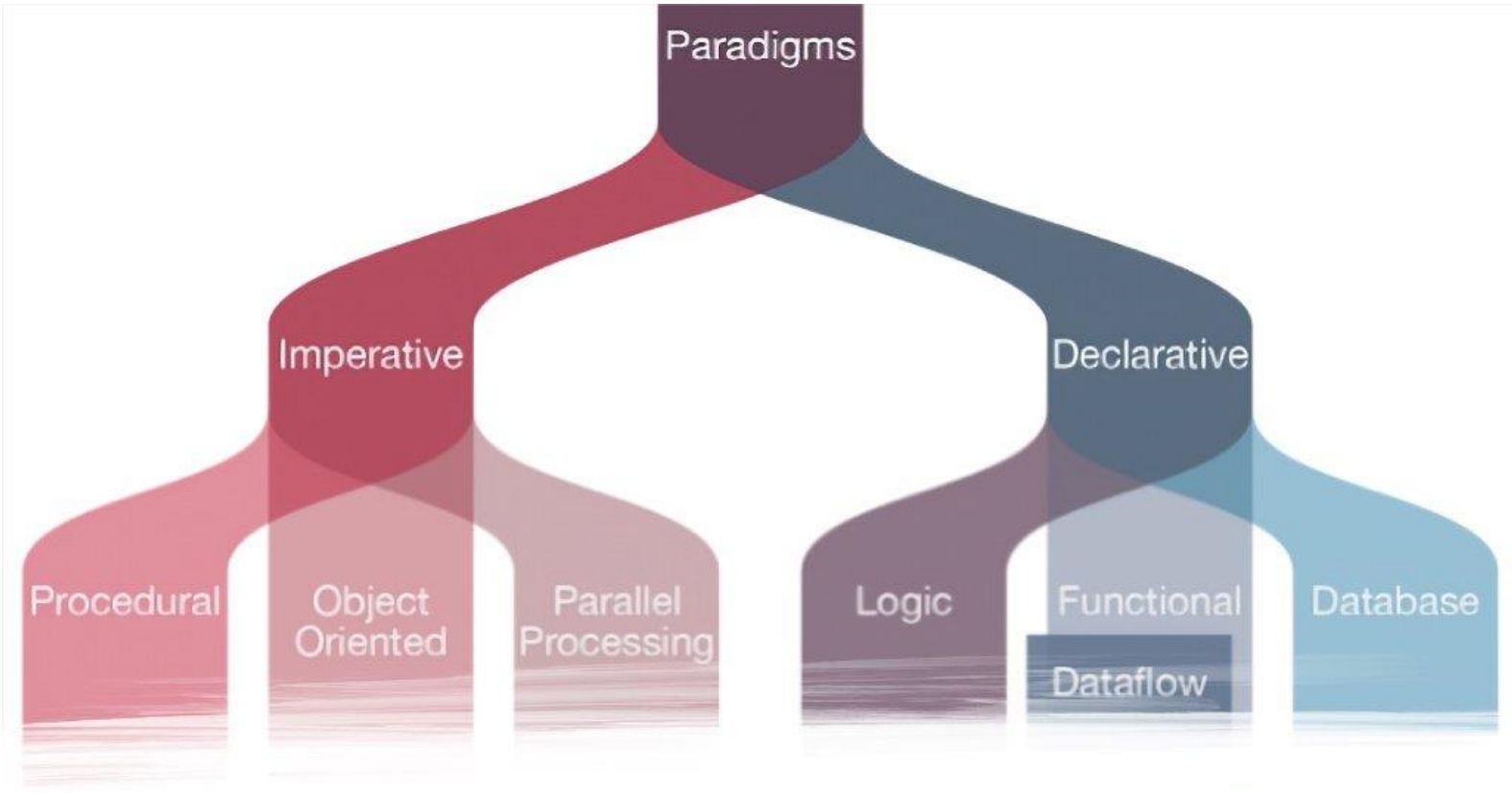
Clean Code: A Handbook of Agile Software Craftsmanship 1st Edition

by Robert C. Martin (Author)

4.4 out of 5 stars 3,247 ratings

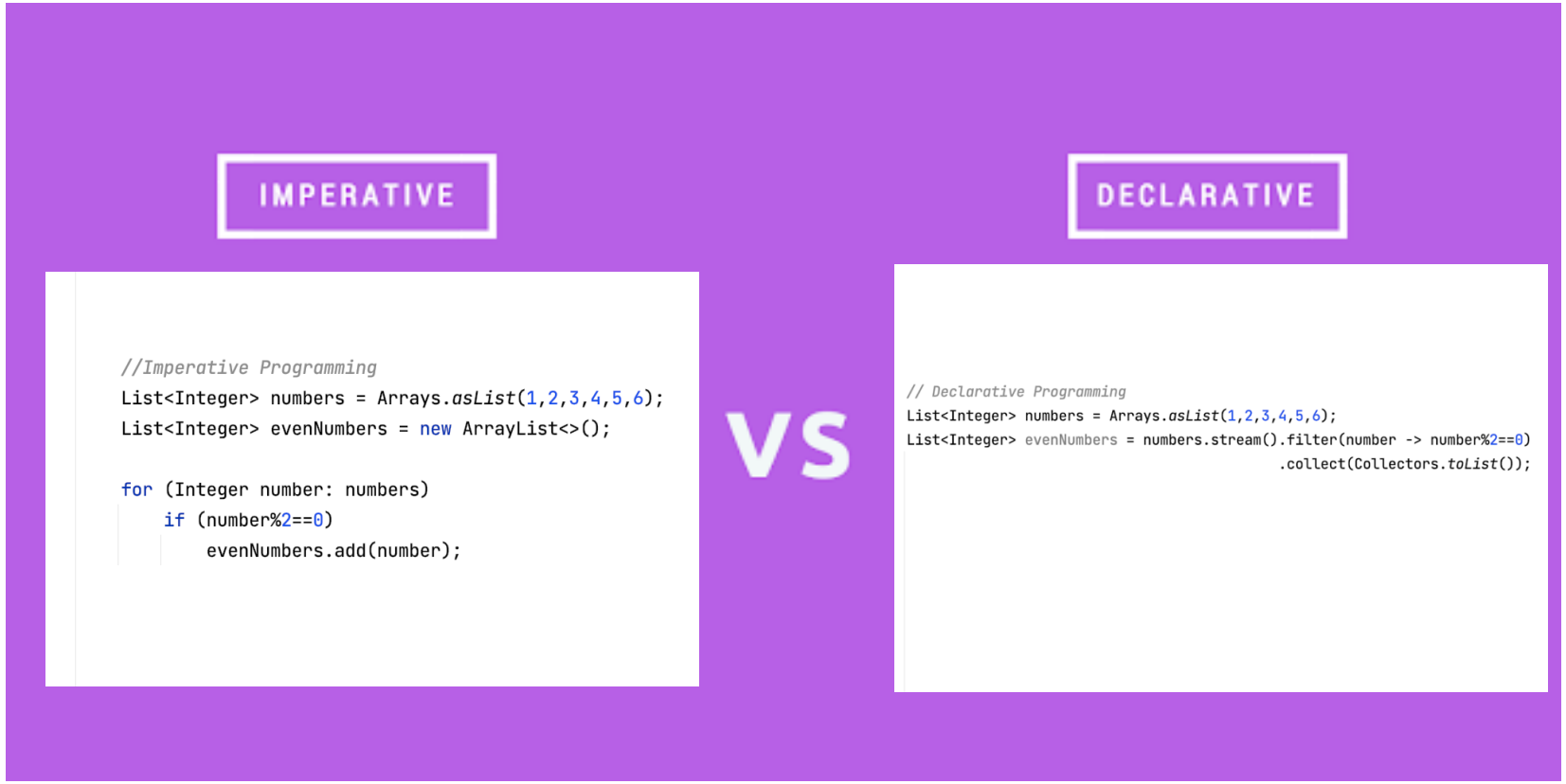


Programming Paradigms



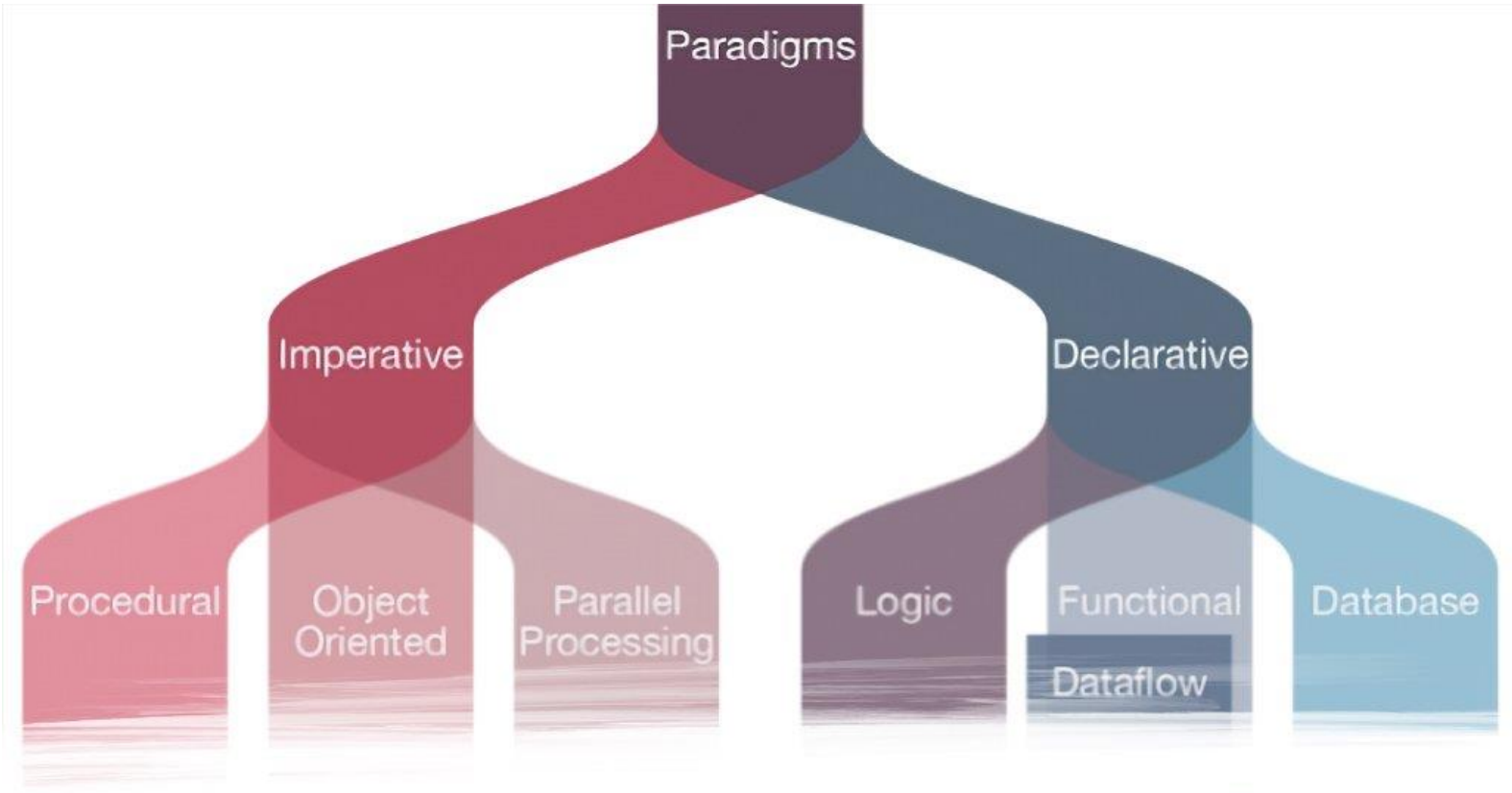
Source: https://www.watelectronics.com/types-of-programming-languages-with-differences/

Programming Paradigms



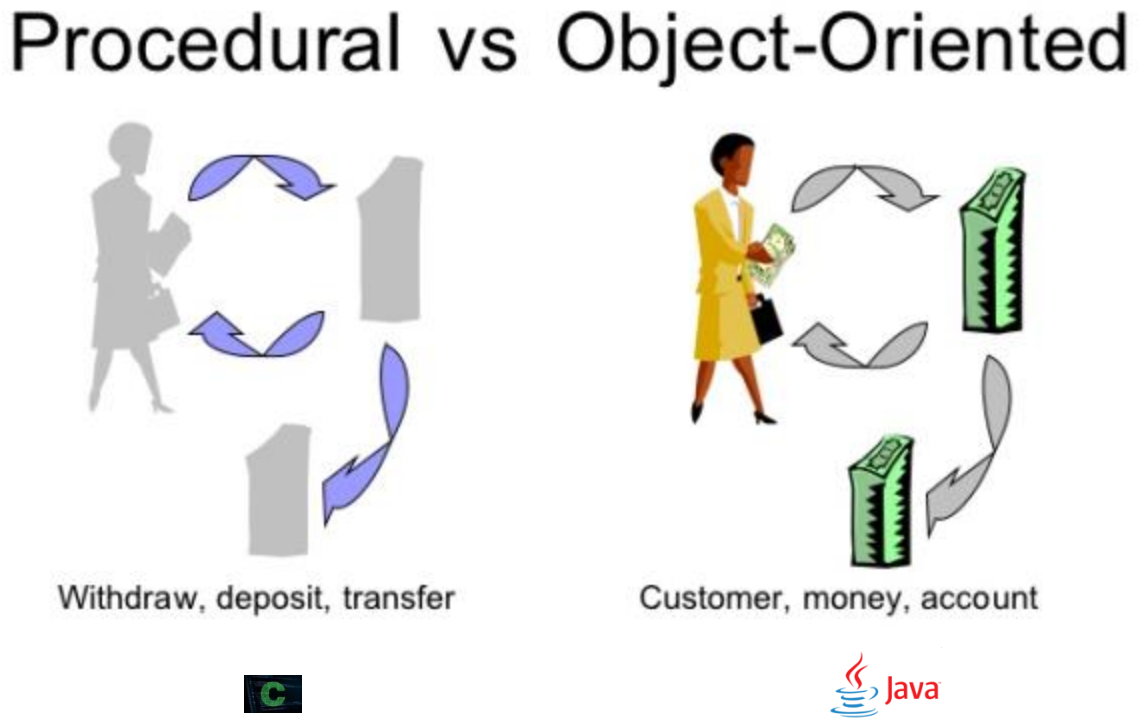
Source. https://medium.com/@vincentbacalso/imperative-vs-declarative-programming-f886d3b65595

Programming Paradigms



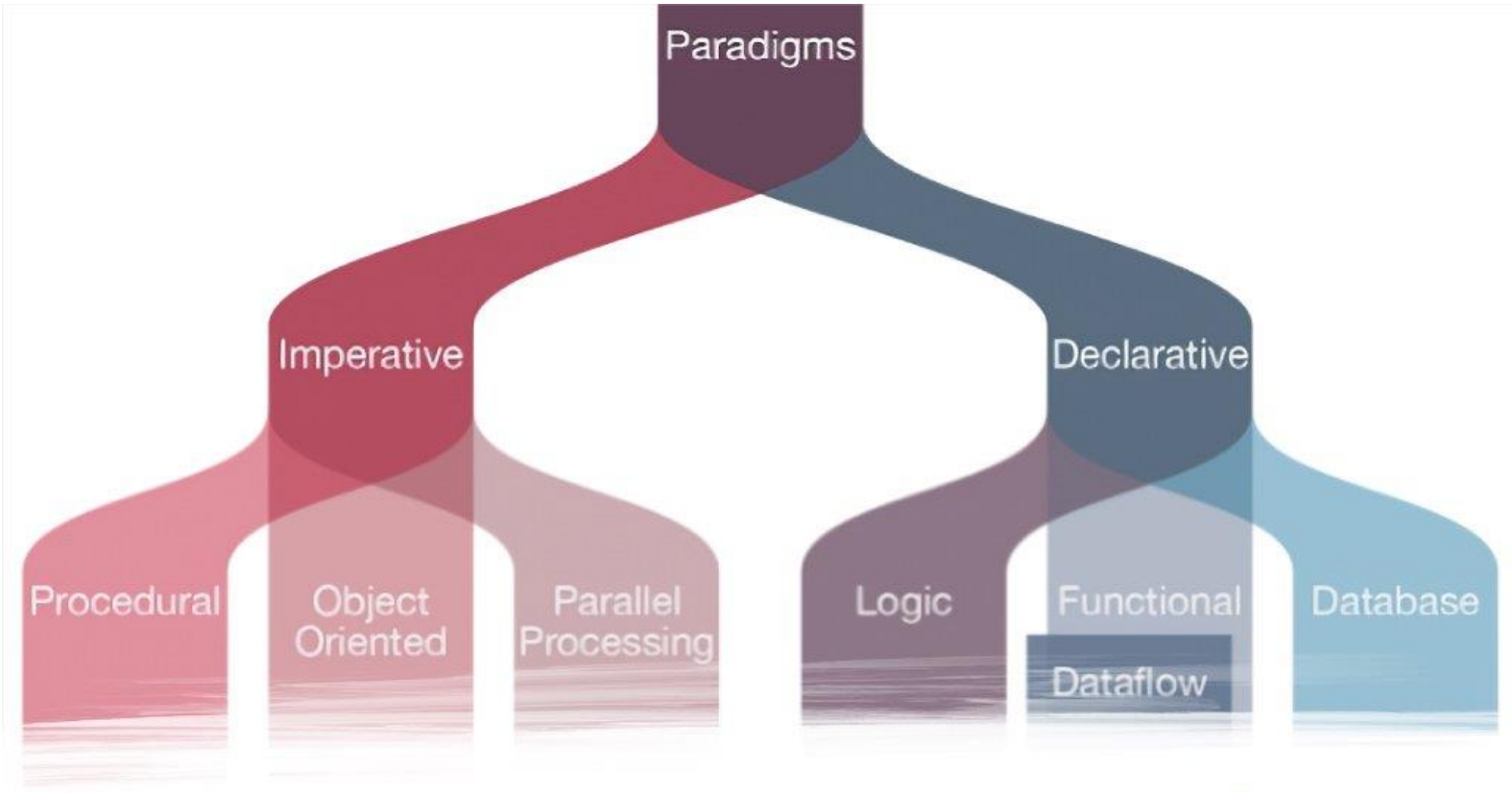
Source: https://www.watelectronics.com/types-of-programming-languages-with-differences/

Programming Paradigms



https://www.alphansotech.com/procedural-vs-object-oriented-programming

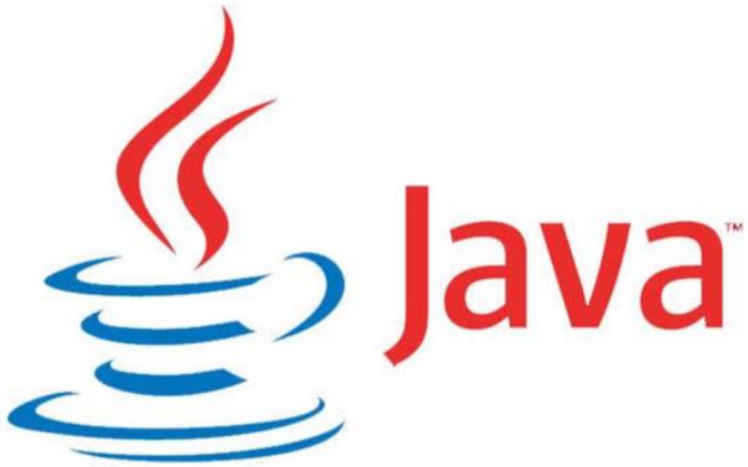
Programming Paradigms



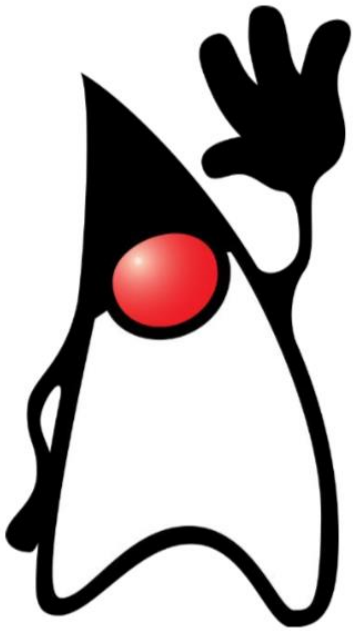
Source: https://www.watelectronics.com/types-of-programming-languages-with-differences/

New IDEs? New Programming Paradigms?

Meet Java!



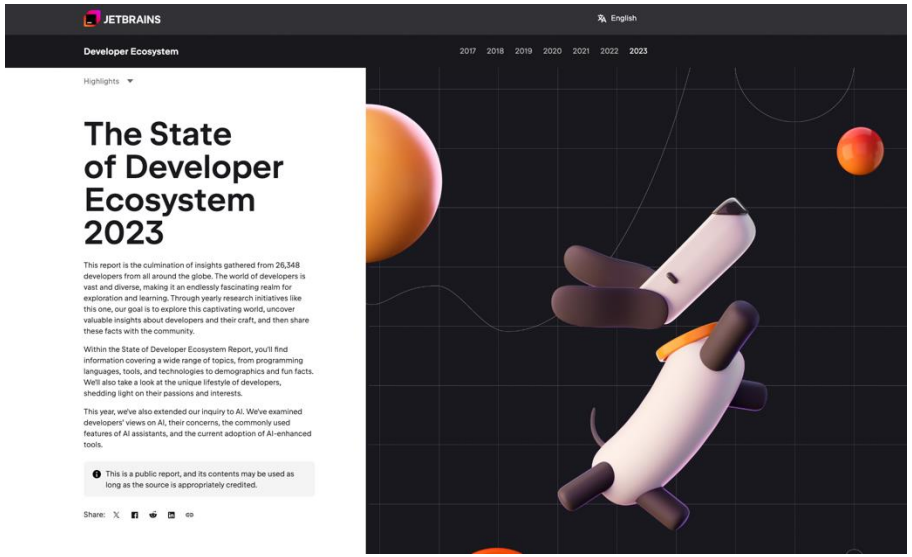
The Java Logo

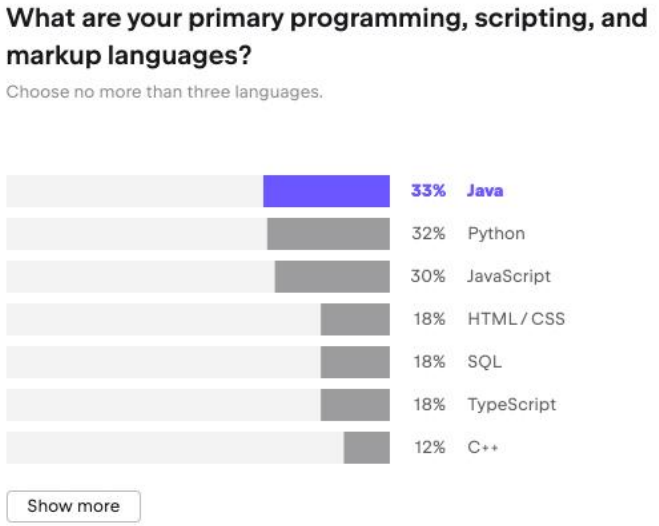


Duke, the Java Mascot

Most Popular Languages

[https://www.jetbrains.com/lp/devecosystem-2023/]





Java is a widely used OOP language

What is Java?

The Java Programming Language (Java)

- An object-oriented programming language.

The Java Virtual Machine (JVM)

- A runtime environment for many platforms.

The Java Standard Library

- A *huge* collection of reusable components.

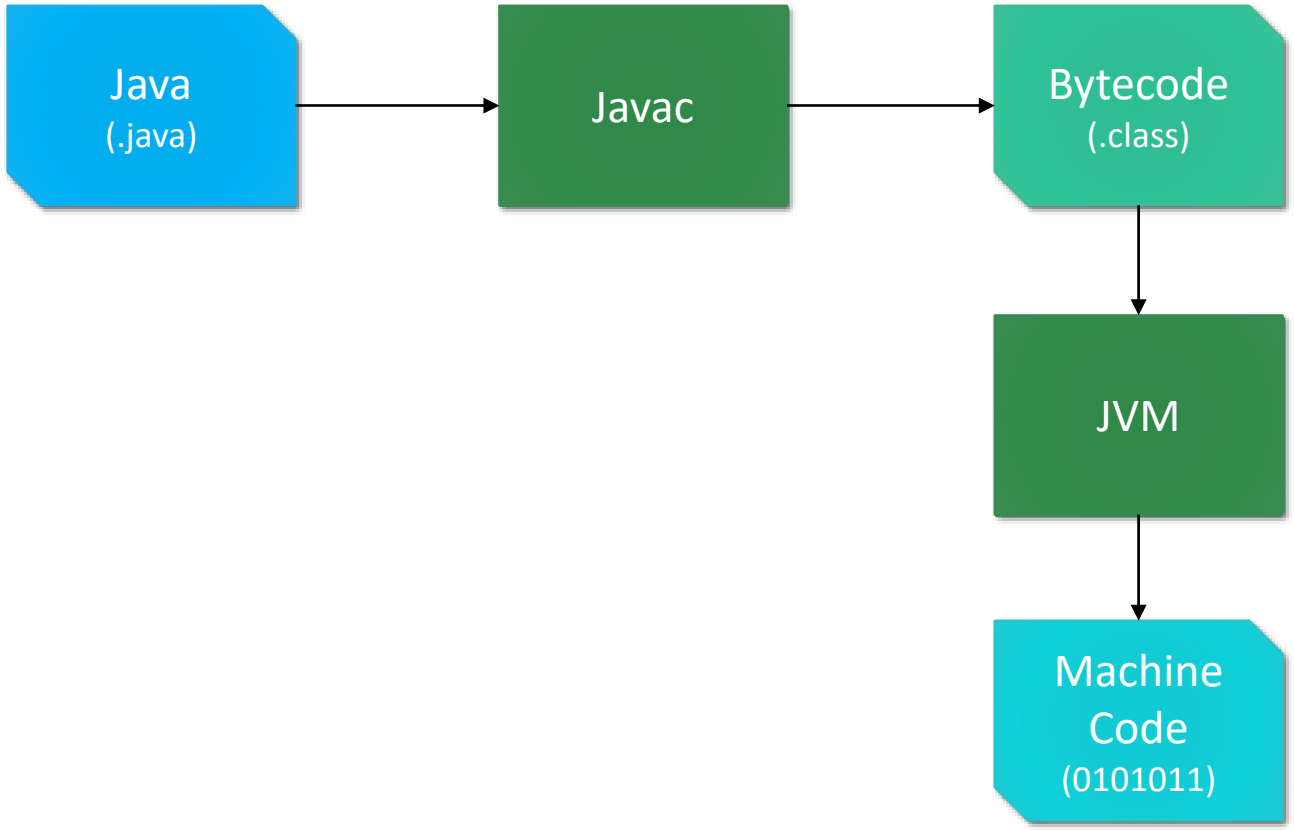
(A large collection of tools around the platform)

The Java Virtual Machine

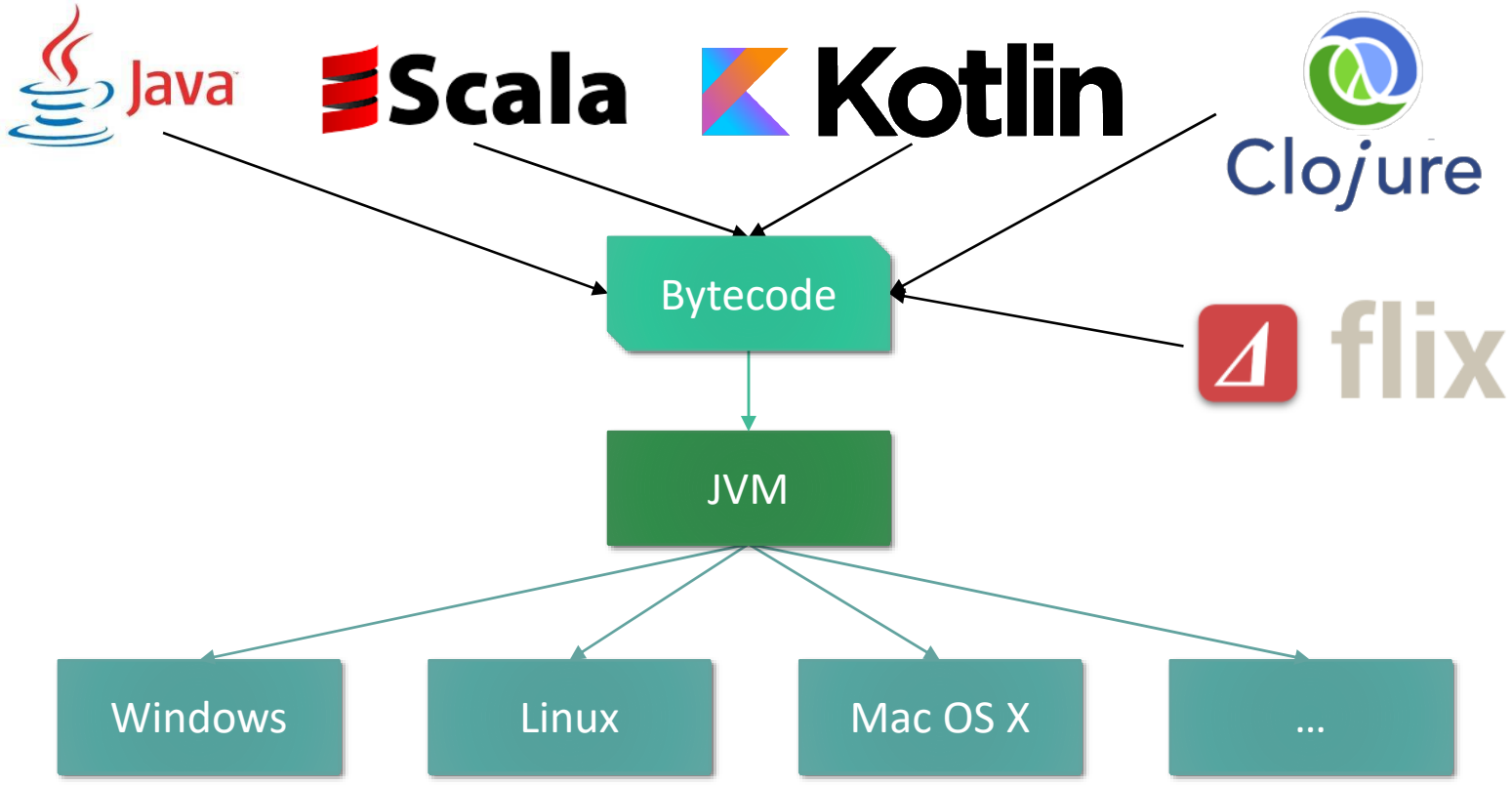
A **runtime environment** that manages your program.

- A safe sandbox; prevents many security issues.
- Allows inspection of your program as it runs via debuggers.
- Compiles your code from JVM bytecode to machine code.

JVM Bytecode

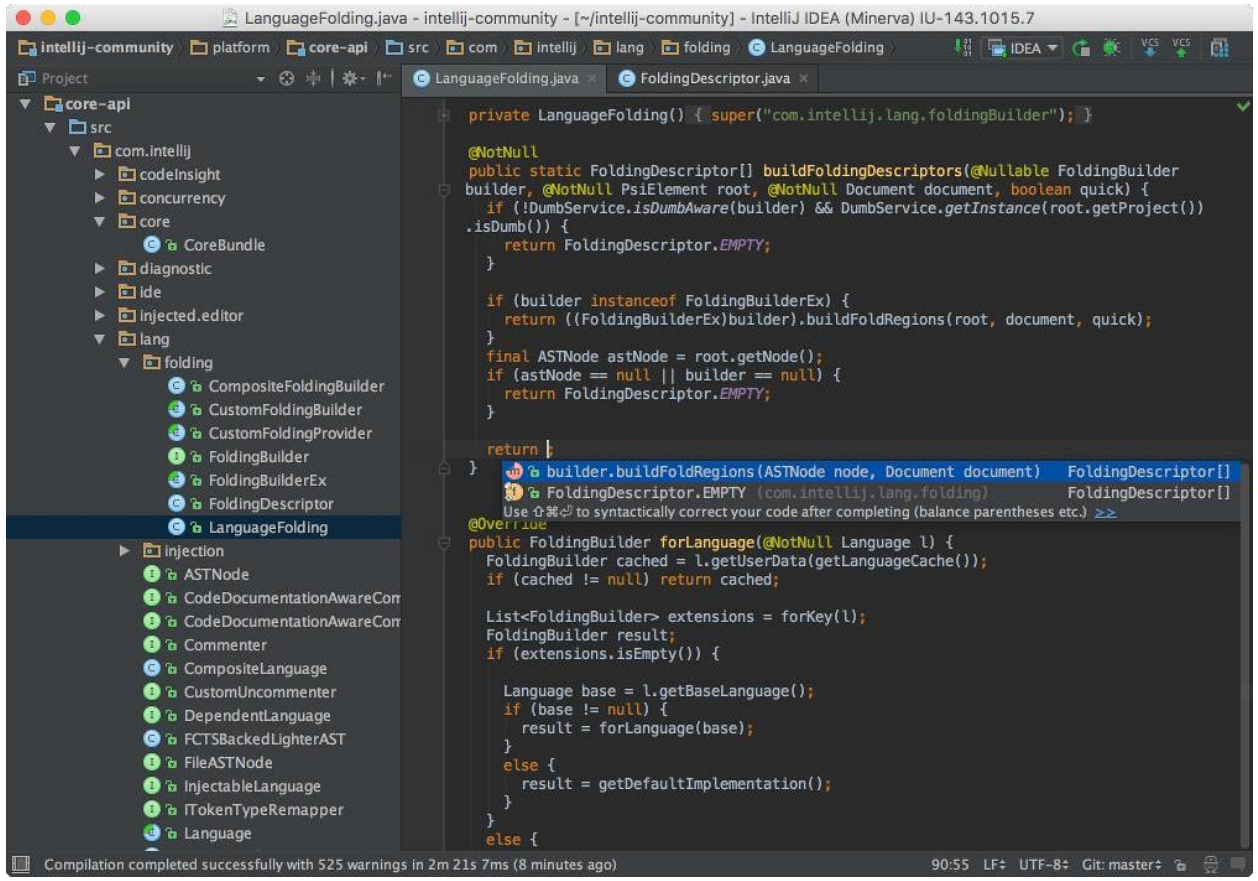


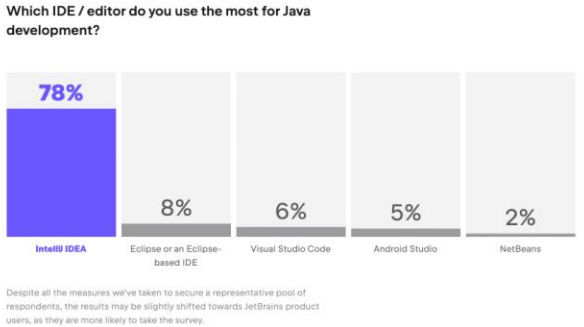
JVM Languages



Integrated Development Environment (IDE)

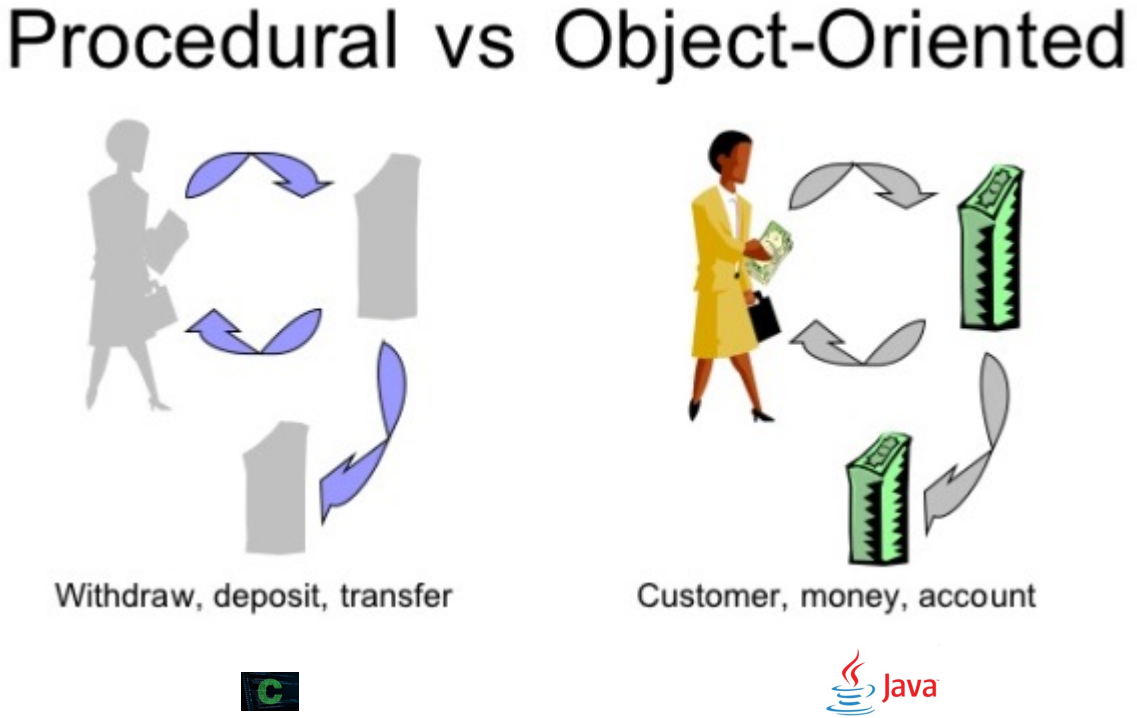
Intellij IDEA





Classes and Objects

Programming Paradigms



https://www.alphansotech.com/procedural-vs-object-oriented-programming

Classes and Objects

A **class** is a template that defines the form of an object.

- It specifies both the ***code*** and ***data*** of an object.
- The ***members*** of a class are: ***fields*** and ***methods.***
  - A field is sometimes called an *instance variable*.
- A class does not exist at run-time (*).

An **object** is a value that exists at run-time.

- An object is an ***instance*** of a class.
- An object is created (allocated) with the `new` keyword.
- An object is automatically destroyed (deallocated) by garbage collection.

(*): At least for the purposes of this course…

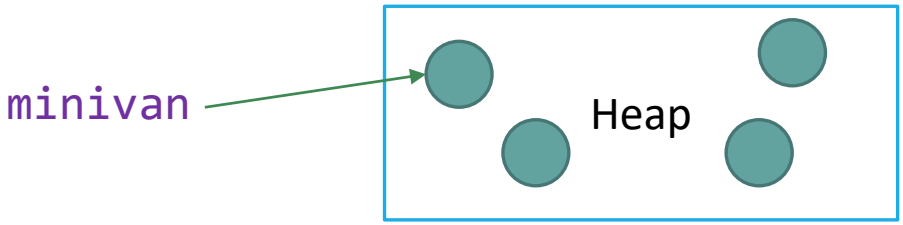
A Prototypical Class

```
class ClassName {

     // fields
     type field1;
     type field2;
     ...

     // methods
     type method1(params...) {
          // method body
     }

     type method2(params...) {
          // method body
     }
     ...

}
```

A Vehicle Class

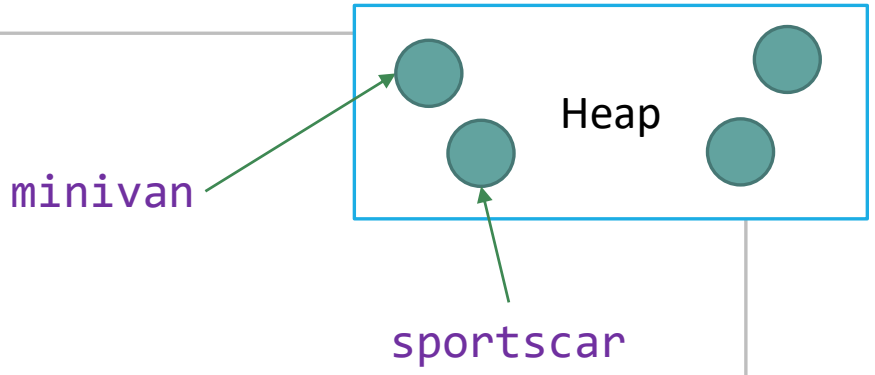
```
class Vehicle {
    int passengers; // number of passengers.
    int fuelcap;    // fuel capacity in gallons.
    int mpg;        // fuel consumption in miles per gallon.
}
```

```
Vehicle minivan = new Vehicle();

minivan.passengers = 7;
minivan.fuelcap    = 16;
minivan.mpg        = 21;
```



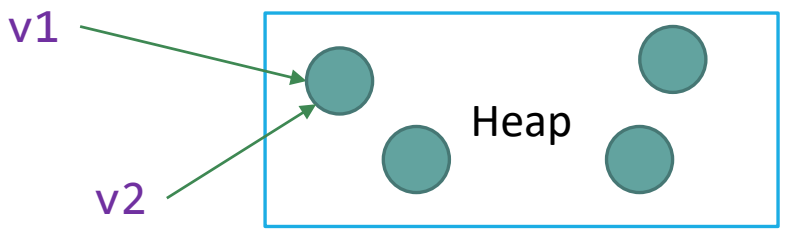
Two Vehicles

```
Vehicle minivan = new Vehicle();
minivan.passengers = 7;
minivan.fuelcap    = 16;
minivan.mpg        = 21;

Vehicle sportscar = new Vehicle();
sportscar.passengers = 2;
sportscar.fuelcap    = 14;
sportscar.mpg        = 12;

int minivanRange = minivan.fuelcap * minivan.mpg;
int sportscarRange = sportscar.fuelcap * sportscar.mpg;

System.out.println("Range of:");
System.out.println("  minivan: " + minivanRange + " miles.");
System.out.println("  sportscar: " + sportscarRange + " miles.");
```



References and Aliasing

```
Vehicle v1 = new Vehicle();
v1.passengers = 7;
v1.fuelcap    = 16;
v1.mpg        = 21;

Vehicle v2 = v1;
v2.passengers = 0;
```



Methods

```
class Vehicle {
    int passengers; // number of passengers.
    int fuelcap;    // fuel capacity in gallons.
    int mpg;        // fuel consumption in miles per gallon.

    double fuelNeeded(int miles) {
        return (double) miles / (double) this.mpg;
    }

     double fuelNeeded2(int miles) {
        return (double) miles / (double) mpg; // equivalent.
    }
}
```

What goes in class?

A class should represent **one logical entity**.

- E.g. a vehicle, a fruit, a tennis player, a ticket, a stadium, …

A class should capture behavior related to its entity.

- E.g. a vehicle has a number of passengers,
- E.g. a fruit has a name, a taste, …,
- E.g. a ticket has a price, a purchase date, …,

A class should have as little responsibility as possible.

- E.g. a vehicle might be responsible for computing fuel consumption, but should not be responsible for computing the distance between two cities.
- If the responsibilities of a class grows to great it should ***delegate***.

Constructors

A ***constructor*** is a special method called when an object is created.

- A class comes with a **default empty constructor.**
- We can use constructors to better control object creation.
- A constructor has the same name as the class name.

```
class Vehicle {
    int passengers; // number of passengers.
    int fuelcap;    // fuel capacity in gallons.
    int mpg;        // fuel consumption in miles per gallon.

    Vehicle(int p, int f, int m) {
        this.passengers = p;
        this.fuelcap = f;
        this.mpg = m;
    }
}
```

Multiple Constructors

A class may have multiple constructors:

```
class Vehicle {
    int passengers; // number of passengers.
    int fuelcap;    // fuel capacity in gallons.
    int mpg;        // fuel consumption in miles per gallon.

    Vehicle(int f, int m) {
        this.passengers = 5;
        this.fuelcap = f;
        this.mpg = m;
    }

    Vehicle(int p, int f, int m) {
        this.passengers = p;
        this.fuelcap = f;
        this.mpg = m;
    }
}
```

Garbage Collection

We allocate objects with

```
Vehicle v1 = new Vehicle(40, 20);
Vehicle v2 = new Vehicle(60, 30);
```

but how are these objects de-allocated?

Through garbage collection!

- The JVM automatically de-allocates objects when they are no longer used.
- An object is no longer used when there are no more live references to it.

Encapsulation

A lovely day at the bank

```
class BankAccount {

    int balance;

    BankAccount(int balance) {
        this.balance = balance;
    }

    boolean withdraw(int amount) {
        if (amount <= this.balance) {
            this.balance = this.balance - amount;
            return true;
        }
        return false;
    }

}
```

A lovely day at the bank

Meet Bob.

```
public class Main {

    public static void main(String[] args){
        BankAccount account = new BankAccount(100);
        account.withdraw(75);

        account.balance = 500_000;
    }

}
```

Bob works for North Korea.

What's the problem?

We want to ensure that our bank accounts satisfy certain **invariants**:

- E.g. an account cannot be overdrawn.
- E.g. a withdrawal is always deducted from the balance.
- E.g. a receipt is always printed when a withdrawal is made.
- …

We want:

To ensure principled access to the state of an object.

How can we do this?

Encapsulation

Key Idea:

- All state of an object should be private (i.e. inaccessible to the outside world.)
- Mutation should only be possible through methods which ensure integrity of the state.

(the state of an object = the value of its fields.)

Example

```
class BankAccount {

    private int balance;

    BankAccount(int balance) {
        this.balance = balance;
    }

    int withdraw(int amount) {
        if (amount <= this.balance) {
            this.balance = this.balance - amount;
            return true;
        }
        return false;
    }

}
```

Access Modifiers

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`<<no modifier>>`	Y	Y	N	N
`private`	Y	N	N	N

Golden rule: *Everything should have the least visibility.*

Corollary: *Everything should be private by default.*

Getters and Setters

```
class Person {
    private String name;
    private int age;

    // constructor omitted.

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}
```

Getters and Setters

```
class Temperature {
    private int celsius;

    // constructor omitted.

    public int getKelvin() {
        return /* ... */;
    }

    public int getCelsius() {
        return /* ... */;
    }

    public int getFahrenheit() {
        return /* ... */;
    }
}
```

What would happen if the program used the celsius field directly?

Packages and Imports

Packages in Java

A **package** is a named collection of classes and interfaces.

- A package name is a dotted name e.g. `foo.bar.baz`.
- The empty name corresponds to the default or root package.

Example I:

- The package `java.lang` contains the class `String`.
- The fully-qualified name of String is `java.lang.String`.

Example II:

- The package `java.net` contains the class `Socket`.
- The fully-qualified name of Socket is `java.net.Socket`.

Why do we want packages?

To distinguish between two concepts that share the same name.

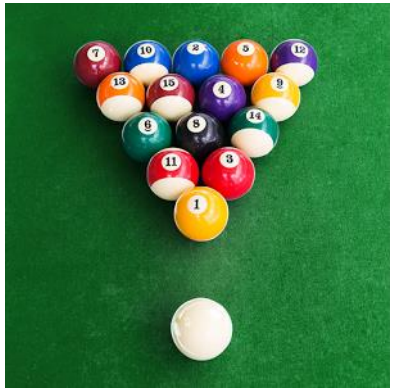
To be able reuse of the same name for different things.

To allow different implementations of the same concept.

To organize a large collection of classes.

```
class Pool
```

```
class Pool
```



The `java.io` Package

Everything to do with input and output:

- `BufferedInputStream`
- `BufferedOutputStream`
- `BufferedReader`
- `BufferedWriter`
- …
- `File`
- …
- `FileReader`
- `FileWriter`
- …

The `java.time` Package

Everything with time

- `Clock`
- `Duration`
- `Instant`
- `LocalDate`
- `LocalDateTime`
- `LocalTime`
- `...`

Choosing Package Names

It is tradition to use **reversed domain names** as package prefixes.

- `org.junit.jupiter.api`
- `org.apache.logging.log4j`
- `ca.uwaterloo.flix`

A simple mechanism to avoid **name clashes**:

- A domain name has an owner.
- A domain name shows affiliation:
  - `dk.aau.cs.yourpackage`
- The syntax of a domain name used to be clear.

Interfaces and Inheritance

A Game of Chess

```
public class ChessPiece {
    private String type;
    private String color;
    private int x;
    private int y;

    // constructor omitted...

    public String getColor() { return color; }
    public int getX() { return x; }
    public int getY() { return y; }

    public boolean move(int newX, int newY) {
        // ...
    }

}
```

Interfaces

An **interface** introduces an **abstraction** for a logical entity.

- An interface consists of a collection of method signatures.
- An interface is **implemented** by one or more classes.
  - Each implementing class must implement the methods specified by the interface.
  - A **class** that implements an interface is a **subtype** of that interface.
- An interface cannot be instantiated by itself.

```
interface ChessPiece {
    String getColor();
    int getX();
    int getY();
    boolean move(int newX, int newY);
}
```

Example

```
class King implements ChessPiece {
    // constructor omitted.
    String getColor() { ... }
    int getX() { ... }
    int getY() { ... }
    boolean move(int newX, int newY) {
        // implement the rules for a King.
    }
}

class Queen implements ChessPiece {
    // constructor omitted.
    String getColor() { ... }
    int getX() { ... }
    int getY() { ... }
    boolean move(int newX, int newY) {
        // implement the rules for a Queen.
    }
}
```

Example

```
public class Main {

    public static void main(String[] args){
        ChessPiece king = new King(“Black”);
        ChessPiece queen = new Queen(“Black”);
        tryMove(king, 1, 1);
        tryMove(queen, 1, 1);
    }

    private static tryMove(ChessPiece piece, int x, int y) {
        if (piece.move(x, y)) {
            System.out.println(“Moved piece successfully!”);
        }
    }

}
```

Example

```
class King implements ChessPiece {
    // constructor omitted.
    String getColor() { ... }
    int getX() { ... }
    int getY() { ... }
    boolean move(int newX, int newY) {
        // implement the rules for a King.
    }
}

class Queen implements ChessPiece {
    // constructor omitted.
    String getColor() { ... }
    int getX() { ... }
    int getY() { ... }
    boolean move(int newX, int newY) {
        // implement the rules for a Queen.
    }
}
```

Duplication!

Class Inheritance

We can also let a class **inherit** from another class:

```
class ChessPiece {
    protected String color;
    protected int x;
    protected int y;

    ChessPiece(String color, int x, int y) {
        // omitted ...
    }

    int getX() { ... }
    int getY() { ... }
    String getColor() { ... }
    boolean move(int newX, int newY) {
      return false;
    }
}
```

Example

```
class King extends ChessPiece {
    ChessPiece(String color, int x, int y) {
        super(color, x, y);
    }

    @Override
    boolean move(int newX, int newY) {
      // implement the rules for a King.
    }
}
```

Abstract Classes

An **abstract class** is a concept between an **interface** and a **class**.

An abstract class can have **fields and methods** (like classes) but it can also have **unimplemented methods** like (like interfaces).

```
abstract class ChessPiece {
    protected String color;
    protected int x;
    protected int y;

    // omitted ...

    abstract boolean move(int newX, int newY);
}
```

Object Oriented Design

Inheritance

A **class** is a type that represents an entity or concept.

- A class has fields and methods.
- The methods have an implementation.

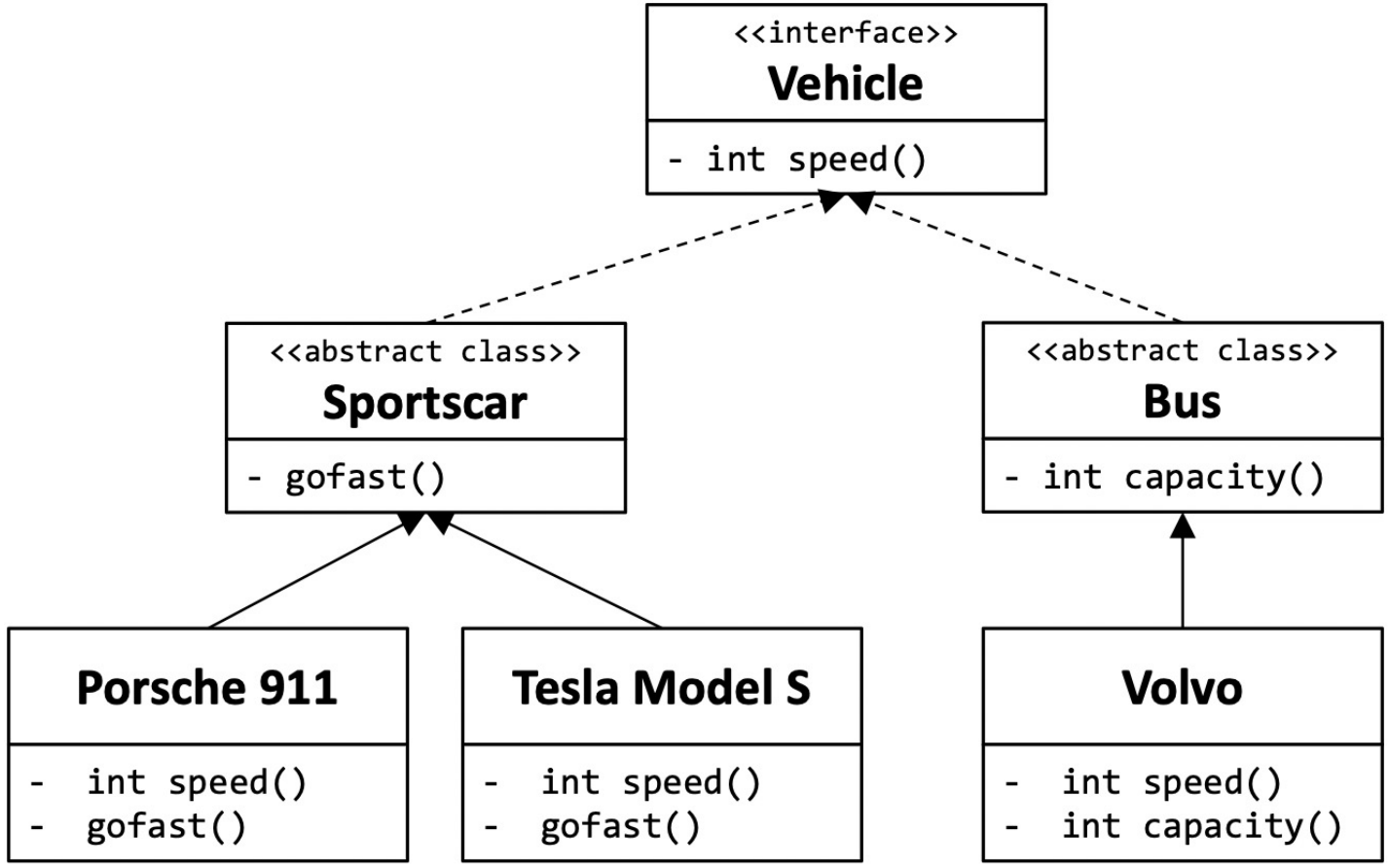
An **interface** is a type that represents an entity or concept.

- An interface has methods, but no fields.
- The methods have no implementation, only a method signature.

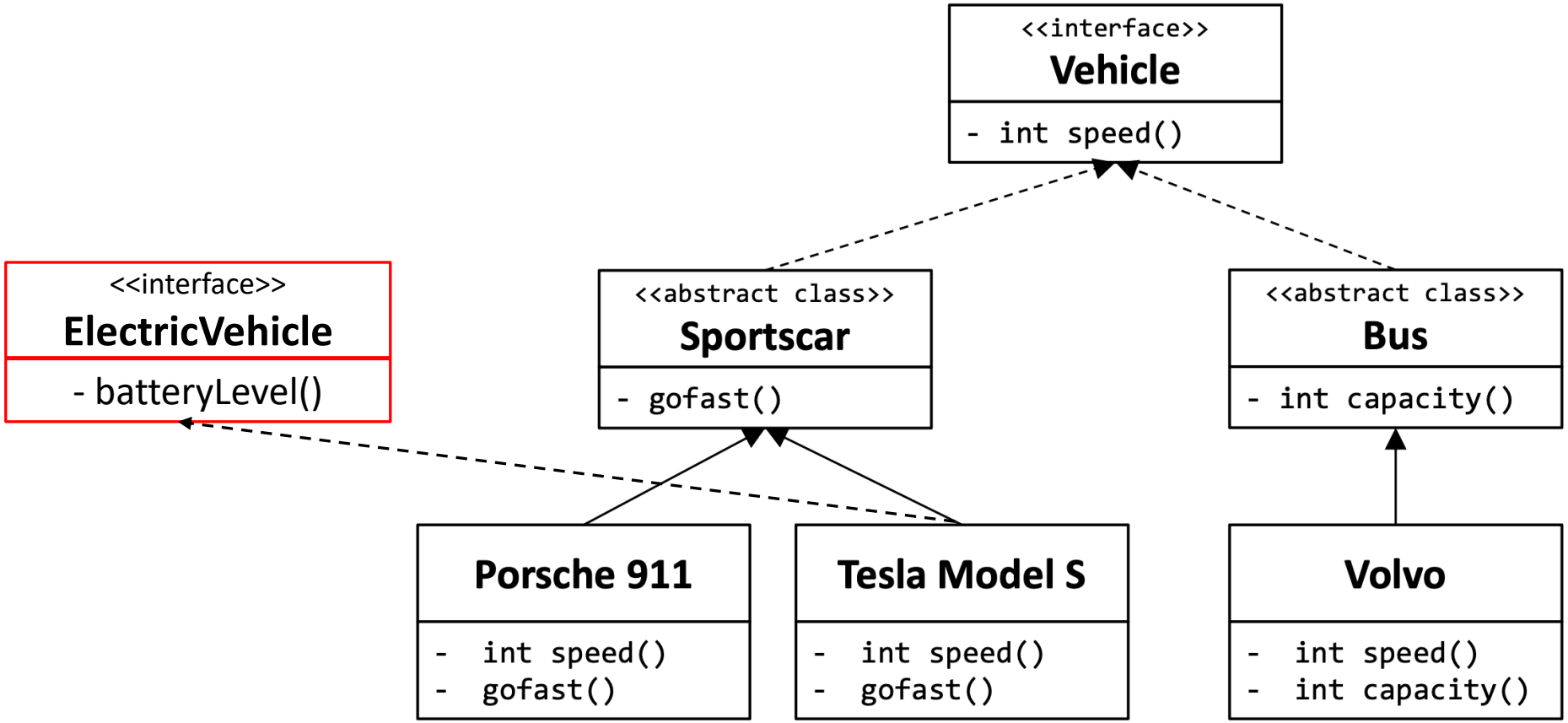
An **abstract class** is a type that represents an entity or concept.

- An abstract class has fields and methods.
- The methods may or may not have an implementation.
- A concept in-between an interface and a class.

Example of OO Design



Multiple Inheritance



What we can do

We have **a lot of flexibility** when creating **class hierarchies**, but:

- We are not allowed to create cycles in the inheritance hierarchy.
- We are not allowed to let an interface extend a class.
- We cannot extend an interface or class marked **final**.
- We cannot override a method marked **final**.

What we should not do

We can use inheritance to create complex hierarchies of interfaces, abstract classes, and classes.

But because we can, does not mean that we should.

Here is a bit of advice:

- Use interfaces.
- Use classes that implement interfaces.
- Avoid deep inheritance.
- Avoid abstract classes.

The java.lang.Object Class

The java.lang.Object Class

Every class extends the Java `Object` class.

Read that sentence again.

This class offers methods that every object is guaranteed to have.

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
protected Object	**clone**() Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	**getClass**() Returns the runtime class of this Object.
int	**hashCode**() Returns a hash code value for the object.
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
String	**toString**() Returns a string representation of the object.
void	**wait**() Causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object.
void	**wait**(long timeout) Causes the current thread to wait until either another thread invokes the **notify()** method or the **notifyAll()** method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos) Causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Equality

We often want to determine when two objects are the same.

But equality is a nebulous concept:

Equality

- If two Person objects have the same first and last name are they the same?
  - What if they have different ages?
- When are two File objects the same?
  - If they have the same filename? The same content? At what point in time?
- When are two date objects the same?
  - If they refer to the same date? In what Calendar? What about leap seconds? Timezones?

Equality in Java

We work with **two types of equality**:

Reference Equality

- Two objects are the same if they refer to the same physical memory location.
- We compare for reference equality with ==.
- Note that: `new Object() != new Object()`.

Object Equality

- Two objects are the same if the equals method **says so**.
- We compare for object equality with `o1.equals(o2)`.
- The default implementation is reference equality.
- The idea is that you override `equals` with your own implementation.
- If so, you must also override `hashCode`. If don't, disaster will follow.

toString

The `toString` method returns a string representation of an object.

```
public static void main(String[] args) {
    Person person = new Person("Magnus");
    System.out.println(person);
}
```

```
Person@4554617c
```

```
@Override
public String toString() {
    return "Person{" + "name='" + name + '\'' + ", age=" + age + '}';
}
```

```
Person{name='Magnus', age=31}
```

Generics

Polymorphism

Poly Morph = Many Shapes

An **object** of **type** `Animal` can be one of several animals, e.g. a `Cat` or a `Giraffe`.

A **method call** `animal.eat()` can have different behavior depending on the runtime type of the animal.

Two Types of Polymorphism

Modern object-oriented languages offer **two types** of polymorphism:

Inheritance (*subtype polymorphism*)

Generics (*parametric polymorphism*)

We need both.

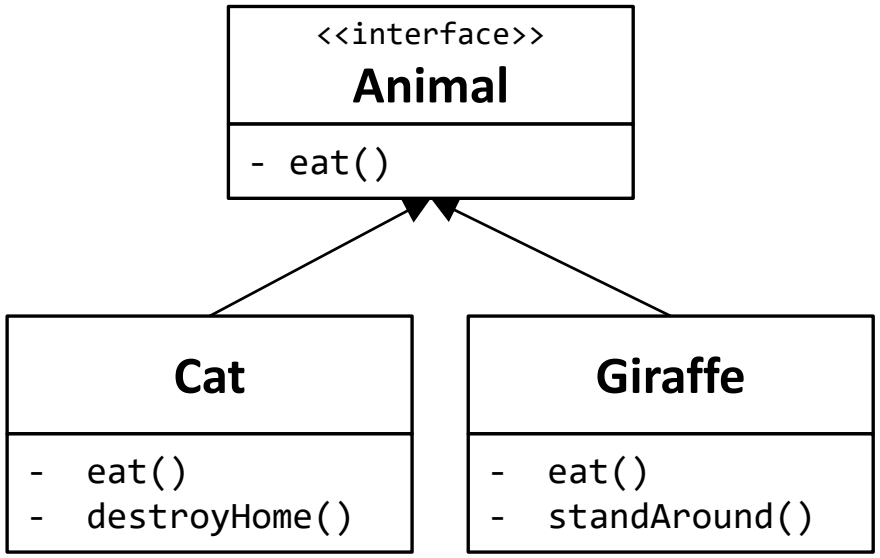
We have already seen subtype polymorphism. Today we shall see generics.

Subtype Polymorphism

Key Idea:

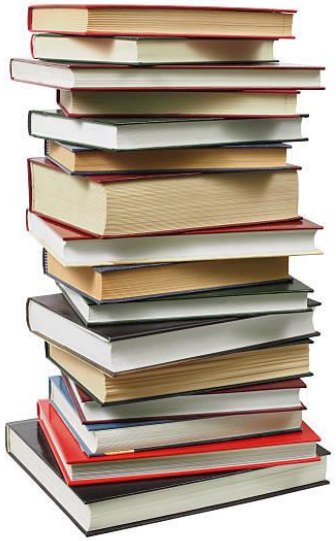
We extract commonalities of several entities into one API.

For example, both a cat and a giraffe eats food. We can create an interface that represents animals and give it an `eat` method. How an animals eat can vary. A cat hunts whereas giraffe forarges.



A Stack of Books

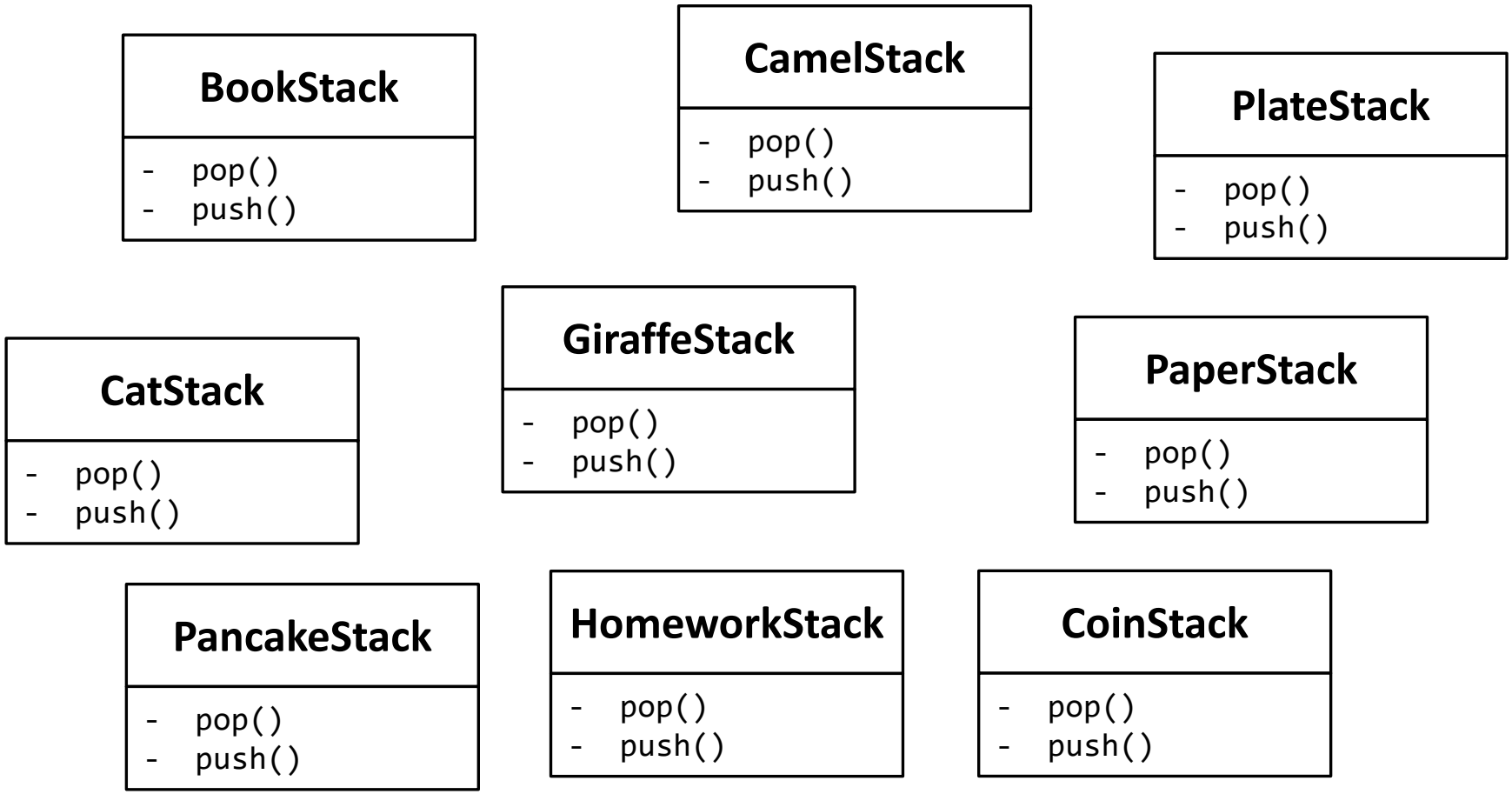
```
public class BookStack {

    private Book[] stack = ...;

    Book pop() {
        // ...
    }

    void push(Book b) {
        // ...
    }

}
```



A Stack of Giraffes

```
public class GiraffeStack {

    private Giraffe[] stack = ...;

    Giraffe pop() {
        // ...
    }

    void push(Giraffe g) {
        // ...
    }

}
```

A Stack of …


BookStack
- pop()
- push()
CamelStack
- pop()
- push()
PlateStack
- pop()
- push()
CatStack
- pop()
- push()
GiraffeStack
- pop()
- push()
PaperStack
- pop()
- push()
PancakeStack
- pop()
- push()
HomeworkStack
- pop()
- push()
CoinStack
- pop()
- push()

Parametric Polymorphism

Key Idea: Introduce a **type parameter**.

We should not have `BookStack` and `GiraffeStack`, but

`Stack<T>`

where T is a type parameter of `Stack`.

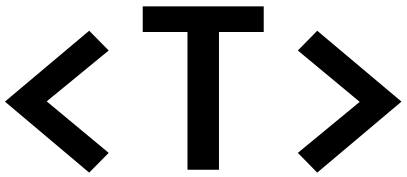
If you want a stack of books you write `Stack<Book>`

If you want a stack of giraffes you write `Stack<Giraffe>`

The types `Stack<Book>` and `Stack<Giraffe>` are different and you cannot accidentally mix them.

Example: Generic Class

```
public class Stack<T> {

    private T[] arr = null;

    T pop() {
        // ...
    }

    void push(T t) {
        // ...
    }

}
```



Collections

The Java Collections Library

A collection is a group of objects organized in some way.

The Java Collections Library is a set of reusable components:

Interfaces

- `Collection<T>, List<T>, Set<T>, Map<K, V>…`

Implementations

- `ArrayList<T>, LinkedList<T>, HashSet<T>, HashMap<K, V> …`

Algorithms

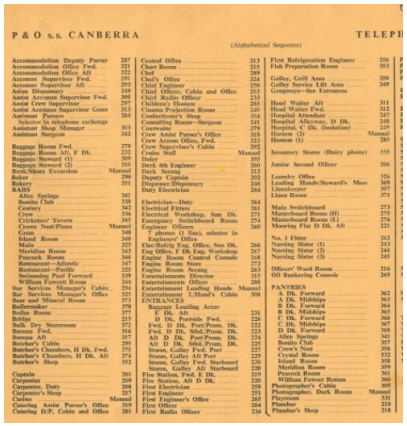
- Finding the minimum or maximum element in a collection.
- Sorting a collection.
- Shuffling a collection.
- …

Examples of Collections

A poker hand (a collection of cards)

A mailbox (a collection of letters)

A telephone directory (a mapping from names to numbers)



Why use Java Collections?

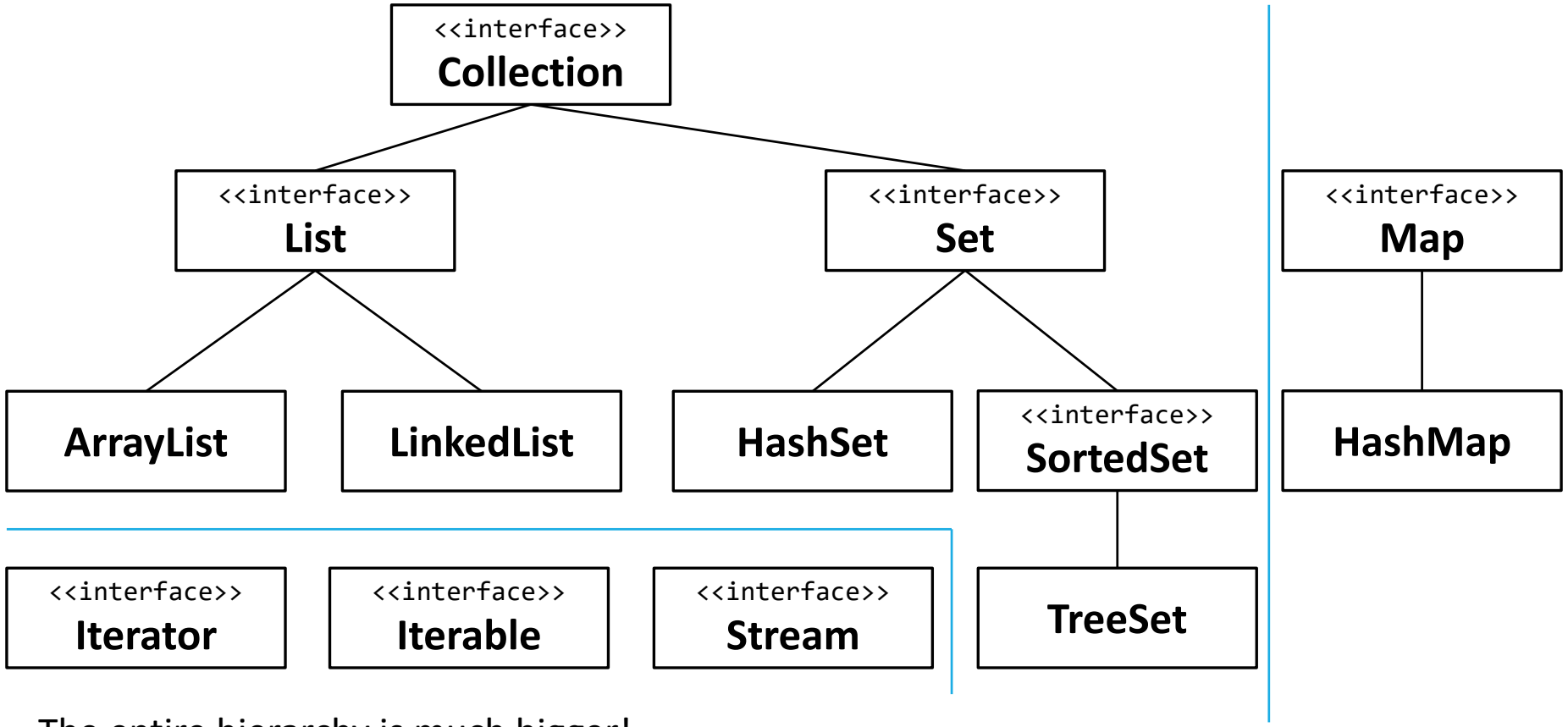
Why should you prefer collections over ordinary arrays?

- Arrays are a low-level abstraction, essentially a chunk of memory.
- Arrays offer few operations and are cumbersome.
- Collections are a high-level abstraction.
- Collections offer rich operations and are easy to use.
- We can save time. We don't have to re-invent the wheel.
- Everyone agrees to use collections. Better for reuse.
- The collections are heavily optimized. Probably better than we could do.

⇒ **Arrays be gone! Use collections from now on!**

(Unless you have good reason not to)

The Java Collections Hierarchy



The entire hierarchy is much bigger!

Overview

`Collection<E>` Defines common methods for all collections.

`List<E>` A sequence of ordered elements.

- The user controls the order of the elements.
- Defines methods to insert and retrieve elements from specific positions.

`Set<E>` An unordered collection of elements without duplicates.

- Relies on the `equals` method to eliminate duplicates.

SortedSet<E> A sorted collection of elements without duplicates.

- Relies on the *natural ordering* of elements or on a comparator.

`Map<K, V>` A mapping between keys of type K and values of type V.

- Relies on the `equals` method to compare keys.

Many other important collections: Stack<E>, Queue<E>, Deque<E>, ...

Collection<E>

Return Type	Method
`boolean`	**`add(E e)`** Ensures that this collection contains the specified element (optional operation).
`void`	**`clear()`** Removes all of the elements from this collection (optional operation).
`boolean`	**`contains(Object o)`** Returns true if this collection contains the specified element.
`boolean`	**`isEmpty()`** Returns true if this collection contains no elements.
`Iterator<E>`	**`iterator()`** Returns an iterator over the elements in this collection.
`boolean`	**`remove(Object o)`** Removes a single instance of the specified element from this collection, if it is present (optional operation).
`int`	**`size()`** Returns the number of elements in this collection.
`Stream<E>`	**`stream()`** Returns a sequential Stream with this collection as its source.

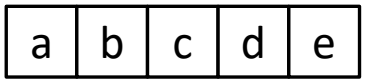
List<E>

Return Type	Method
`void`	**`add(int index, E element)`** Inserts the specified element at the specified position in this list (optional operation).
`E`	**`get(int index)`** Returns the element at the specified position in this list.
`int`	**`indexOf(Object o)`** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`int`	**`lastIndexOf(Object o)`** Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E`	**`remove(int index)`** Removes the element at the specified position in this list (optional operation).
`List<E>`	**`subList(int fromIndex, int toIndex)`** Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

ArrayList<E> and LinkedList<E>

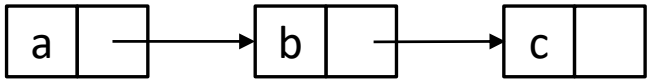
`ArrayList<E>` A list implementation backed by an array.

- Good random access performance.
  - Quick to find an element given an index.
- Cannot insert elements in the front.



`LinkedList<E>` A list implementation backed by linked cells.

- Bad random access performance.
  - Slow to find an element given an index, must scan through the list.
- Can insert elements in the front.



Most common to use `ArrayList<E>`.

Example: List<E>

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

    List<String> cards = new ArrayList<>();
    cards.add("Ace of Hearts");
    cards.add("Queen of Spades");
    cards.add("Ten of Hearts");
    cards.add("Ace of Hearts");
    cards.add("Two of Clubs");

    cards.remove(1);
    int index = cards.indexOf("Ten of Hearts");
    System.out.println(index);

}
```

Collections

Return Type	Method
`static int`	**`binarySearch(List<T> list, T key)`** Searches the specified list for the specified object using the binary search algorithm.
`static int`	**`binarySearch(List<T> list, T key, Comparator<T> c)`** Searches the specified list for the specified object using the binary search algorithm.
`static void`	**`shuffle(List<T> list)`** Randomly permutes the specified list using a default source of randomness.
`static void`	**`sort(List<T> list)`** Sorts the specified list into ascending order, according to the natural ordering of its elements.
`static void`	**`sort(List<T> list, Comparator<T> c)`** Sorts the specified list according to the order induced by the specified comparator.

(Note: I simplified some of the types for readability.)

Comparator<T>

```
public interface Comparator<T> {

    /**
     * Compares its two arguments for order.
     *
     * Returns a negative integer, zero, or a positive integer
     * as the first argument is less than, equal to, or greater
     * than the second.
     */
    int compare(T o1, T o2);
}
```

Example: Set<E>

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

    Set<String> cards = new HashSet<>();
    cards.add("Ace of Hearts");
    cards.add("Ace of Hearts");
    cards.add("Queen of Spades");
    cards.add("Queen of Spades");
    cards.add("Ten of Hearts");
    cards.add("Two of Clubs");

    cards.remove("Ten of Hearts");
    int size = cards.size();
    System.out.println(size);

}
```

Map<K, V>

Return Type	Method
boolean	**containsKey(Object key)** Returns true if this map contains a mapping for the specified key.
boolean	**containsValue(Object value)** Returns true if this map maps one or more keys to the specified value.
V	**get(Object key)** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V	**getOrDefault(Object key, V defaultValue)** Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
Set<K>	**keySet()** Returns a Set view of the keys contained in this map.
V	**put(K key, V value)** Associates the specified value with the specified key in this map (optional operation).
Collection<V>	**values()** Returns a Collection view of the values contained in this map.

Example: Map

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

    Map<String, Integer> birthYear = new HashMap<>();
    birthYear.put("George Washington", 1732);
    birthYear.put("Abraham Lincoln", 1809);
    birthYear.put("Barack Obama", 1961);

    Integer obamaYear = birthYear.get("Barack Obama");
    System.out.println("Mr. Obama was born in: " + obamaYear);

    System.out.println("We have information on the presidents:");
    for (String president : birthYear.keySet()) {
        System.out.println("Mr. " + president);
    }

}
```

Software Testing

What is a test?

An **experiment** to determine the outcome of a program execution.

- A test is **passed** if the outcome matches our expectations.
- A test is **failed** if the outcome differs from our expectations.

We can test a program when:

- The program compiles and runs.
- The expected outcome is understood.

Note: We can test an *incomplete* program as long as it compiles and runs!

Why care about testing?

As a technique to ensure correctness.

As a method to implement tricky or hard-to-understand algorithms.

As a mechanism to enable change and refactoring without causing breakage.

My Initial View

As a technique to ensure correctness.

As a method to implement tricky or hard-to-understand algorithms.

As a mechanism to enable change and refactoring without causing breakage.

My Changed View

As a technique to ensure correctness.

As a method to implement tricky or hard-to-understand algorithms.

As a mechanism to enable change and refactoring without causing breakage.

Technical Debt

Every codebase has an amount of *technical debt.*

You increase the technical debt:

- Every time you add features without documentation or tests.
- Every time you add a hack without refactoring it afterwards.
- Every time a feature must be ready “by Monday”.
- Every time a new version of a language is released and you do not update.
- Every time a new version of a library is released and you do not update.
- Every time a new version of s platform is released and you do not update.

Software Rot

As the technical debt increases the codebase literally rots!

What can be done to decrease the technical debt?

- Continuously improve and maintain the codebase through refactoring.
- Continuously update to newest versions of languages, libraries, platforms.
- **Add unit and system tests to ensure the correctness and maintainability.**

A codebase may incur so much technical debt that it is cheaper to throw it away and start from scratch than it is to fix it!

- Happens all the time in software development, but not good!

Dimensions of Testing

Question	Dimensions	
How are tests run?	Manual	Automated
What does a test cover?	Unit Test	System Test
What assumptions does the test make about the implementation?	Black Box	White Box
Is the test written before or after the implementation is written?	Test First	Test Later

Every decision involves different trade-offs.

Manual vs. Automated Testing

Manual Testing: A human runs the program and inspects the result.

- Advantages: works well for some scenarios, e.g. UI testing.
- Disadvantages: tedious, time consuming, and error-prone.

Automated Testing: A computer runs the program and compares the result to a known expected result.

- Advantages: tests are repeatable, little manual effort.
- Neutral: requires time in the short run, but pays off in the long run.
- Disadvantages: hard to test certain requirements.

Unit Test vs. System Test

Unit testing involves testing individual components in isolation.

- A unit test might test **a single method or a few methods of a class.**

System testing involves testing larger components in cooperation.

- A system test might test **numerous classes working together.**

A good test suite should have both unit and system tests.

- Both are necessary and serve different purposes.
- Unit testing is easy. System testing is hard.
- Big Companies invest Big Bucks in good system tests. (Think Google, …)

Introduction to jUnit

A Java Unit Testing Framework

- Extremely popular and widely-used.
- Integrated into Intellij IDEA.
- Easy to use and get started with.
- Annotation based.

Asserts in jUnit

Annotation	Meaning
`assertTrue`	Asserts that the given value is true.
`assertFalse`	Asserts that the given value is false.
`assertNull`	Asserts that the given value is null.
`assertNotNull`	Asserts that the given value is not null.
`assertEquals`	Asserts that the two values are equal.
`assertArrayEquals`	Asserts that the two arrays are equal.
`fail`	Unconditional fails the current unit test.

Examples

```
@Test
public void isOverdrawn01() {
    BankAccount account = new BankAccount(100);
    account.withdraw(500_000);
    assertTrue(account.isOverdrawn());
}

@Test
public void isOverdrawn02() {
    BankAccount account = new BankAccount(100);
    account.withdraw(25);
    assertFalse(account.isOverdrawn());
}
```

Annotations

Annotation	Meaning
`@Test`	Marks the method as a unit test.
`@BeforeAll`	Marks that a method should run before all unit tests are run.
`@BeforeEach`	Marks that a method should run before each unit test.
`@AfterEach`	Marks that a method should run after each unit test has run.
`@AfterAll`	Marks that a method should run after all unit tests have run.
`@Disabled`	Marks that a test should be skipped.

Example: BeforeEach

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class TestBankAccount {

    BankAccount account = null;

    @BeforeEach
    public void beforeEach() {
        account = new BankAccount(100);
    }

    @Test
    public void withdraw01() {
        account.withdraw(25);
        assertEquals(75, account.getBalance());
    }

}
```

Black Box vs. White Box

A **white box test** has knowledge of the unit under test (UUT).

- Easy to know what corner cases to test.
- Easy to know when we have tested sufficiently.
- Tests *may not* be faithful to the specification.
- Tests *may not* sufficiently test a *different* implementation.
- Tests and implementation are coupled.

A **black box test** has *no* knowledge of the unit under test.

- Relies on the specification and only the specification.
- Avoids any assumptions about the implementation.
- A more paranoid and adversarial approach.
- Red Team – Blue Team.

Test Coverage

Q: How do determine if a test suite is useful?

A: We measure the *test coverage* of a test suite:

- **Method Coverage:** the percentage of methods executed.
- **Statement Coverage:** the percentage of statements executed.
- **Path Coverage:** the number of paths taken.

- We have great IDE support for this!

Ensuring high test coverage is a measurable and achievable goal, but…

Production and Test Code

We should keep **production** and **test code** completely separate.

- Production code must never depend on the test code!
- A common strategy is to separate the code into two directories:
  - Production code goes in src/
  - Test code goes in test/

As a general rule the test code is *not* part of any release.

- Test code may contains confidential information (e.g. database credentials).
- Test code may contain insecure components (e.g. security overrides).
- Counter-point: Open source *libraries* typically do ship their tests.

Assertions

We use unit tests and assertions to check for correctness.

- We do not ship these tests to the customer (see previous slide).

We can also add assertions directly to the product code.

- Advantages: We immediately discover abnormal situations during program execution and the program is (likely) shutdown before it does any harm.
- Disadvantages: We must pay a performance cost for these checks.

```
public void deposit(int amount) {
    assert amount >= 0 : “Non-positive amount”
    ...
}
```

Contracts

Pre and Post Conditions

- A **pre condition** is a boolean expression which must be true immediately before execution of a method.
- A **post condition** is boolean expression which must be true immediately after execution of a method.

A **contract** is pre and post condition of a method.

- It is a formal way to state that “if you promise X then I will promise Y”.

```
public void deposit(int amount) {
    assert amount >= 0 : “Pre-condition failed.”
    ...
    assert balance == oldBalance + amount : “Post-condition failed.”
}
```

It is impossible to test X!

"If a feature cannot be tested
it simply does not exist."

Dijkstra on Testing

"Testing cannot be used to show the absence of bugs, only to show their presence."

We can always write one more test. We will never be done!

When should tests be written?

Before writing the implementation:

- Helps explore the problem domain.
- Helps design the appropriate methods and interfaces.
- Helps drive the implementation.
- Ensures we have tests when we are done.

After writing the implementation:

- We have better knowledge of the problem domain.
- Easier to determine what tests to write.
- Quicker to write those tests.
- Risk of saying “we will add tests later” (never happens.)

Test-Driven Development

A method, not for testing, but for software development.

Key Idea:

Use unit tests to drive the implementation!

Added bonus: You get unit tests for free.

The TDD Rhythm

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a small change.
4. Re-run all tests and see them succeed.
5. Refactor the code.
6. (Re-run all tests.)

The Three Laws of TDD

According to Robert Martin “Uncle Bob”:

First Law: You may not write production code until you have written a failing unit test.

Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Third Law: You may not write more production code than is sufficient to pass the currently failing test.

According to Bob, if we do this: We will write dozens of tests every day, hundreds of tests every month, and thousands of tests every year.

F.I.R.S.T. Principles

Fast

- Tests should run fast so that you can run them often.

Independent

- Tests should be independent to ensure correctness and ease of debugging.

Repeatable

- Tests should run on any environment, including your laptop and build server.

Self-Validating

- Every test should have a clear boolean output: Test failed for this reason.

Timely

- Tests should be written before writing the production code.

Error Handling with Exceptions

How can programs go wrong?

The program is used incorrectly:

- User wants to open a file, but the file does not exist.
- User wants to connect to a domain, but the domain does not exist.
- User wants to perform an action, but does not have permission.

The program depends on unreliable resources:

- The network is down.
- The file system is full.
- No more memory is available.

The program violates a protocol or performs nonsensical operations:

- Allocated memory is never deallocated.
- Deallocated memory is still accessed.
- Tries to divide by zero.

How can we handle errors?

Incorrect use of the program:

- Explain to the user what is wrong and ask for a corrective action!
- Recoverable, we can ask the user for help.

Unreliable resources:

- If the network is down, we can **try again later**.
- If the file system is full, we can **ask for corrective action**.
- Often recoverable. Perhaps the problem will fix itself or the user can help.

Protocol Violations

- Programming bugs: we, as the developers, made a mistake.
- Almost always unrecoverable, the program itself must be **corrected**.
- Not much to do, other than to try to save data, and exit the program.

How can we discover an error?

We need a **mechanism** to report when something has gone wrong.

Typical scenario: We call a function/method/procedure. Did it go well?

- **A Global Variable holds an error code, if any.**
  - Commonly used in C. Old school. Avoid.
- **Return an error code, if any. Otherwise return a success value.**
  - Commonly used in C, but also in many other languages.
  - What if we actually wanted to return a value? Sometimes -1 for error, and rest for the result.
- **Return a data structure which encodes success/failure.**
  - Commonly used in functional languages, e.g. Scala, Haskell, Ocaml, etc.
- **Redirect control-flow with an *exception*.**
  - Commonly used in Java, C#, and current industry languages.
  - What we will talk about for the remainder.

Exceptions

A control-flow mechanism to handle abnormal situations.

- We **throw** and **catch** exceptions.
- An exception is an object (an instance of a class).

We **throw** an exception when we want to abort execution in a method.

- A throw statement immediately abort execution, similar to a return.
- A return statement only exits the *current* method, but a throw statement *continues* exiting methods until the exception is caught.

We **catch** an exception when we want to handle it.

- Execution continues from where the exception is caught.
- If an exception is never caught, it reaches main, and the program aborts.
- The part that catches an exception is called an *exception handler*.

Throwing Exceptions

```
public class BankAccount {

    private int balance;

    public void withdraw(int amount) {
        if (amount <= 0) {
            String msg = "Amount must be positive.";
            throw new IllegalArgumentException(msg);
        }

        if (amount >= balance) {
            String msg = "Amount must be less than balance.";
            throw new IllegalArgumentException(msg);
        }

        balance = balance - amount;
    }

}
```

Try-Catch

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    BankAccount account = new BankAccount(100);

    while (true) {
        System.out.println("Enter an amount to withdraw:");
        int amount = scanner.nextInt();

        try {
            account.withdraw(amount);
        } catch (IllegalArgumentException e) {
            System.out.println(“Something went wrong.");
        }
    }
}
```

Custom Exceptions

```
public class NegativeAmountException extends RuntimeException {

    public NegativeAmountException() { }

}

public class InsufficientFundsException extends RuntimeException {

    private int deficit;

    public InsufficientFundsException(int deficit) {
        this.deficit = deficit;
    }

    public int getDeficit() {
        return deficit;
    }

}
```

Throwing Exceptions

```
public class BankAccount {

    private int balance;

    public void withdraw(int amount) {
        if (amount <= 0) {
            throw new NegativeAmountException();
        }

        if (amount >= balance) {
            int deficit = amount - balance;
            throw new InsufficientFundsException(deficit);
        }

        balance = balance - amount;
    }

}
```

Catching Multiple Exceptions

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    BankAccount account = new BankAccount(100);

    while (true) {
        System.out.println("Enter an amount to withdraw:");
        int amount = scanner.nextInt();

        try {
            account.withdraw(amount);
        } catch (NegativeAmountException e) {
            System.out.println("Amount cannot be negative!");
        } catch (InsufficientFundsException e) {
            System.out.println("You are short: " + e.getDeficit());
        }
    }
}
```

Uncaught Exceptions

Uncaught exceptions propagate to **main** and abort the program.

Every exception includes a **stack trace** of the methods that were currently on the call stack when the exception occurred.

```
Exception in thread "main" NegativeAmountException
     at BankAccount.withdraw(BankAccount.java:11)
     at Main.main(Main.java:13)
```

Real stack traces are typically much longer!

- Try a Google search for “long stack trace”.

Checked and Unchecked

In Java exceptions come in two variants:

- A **checked** exception must be caught or re-thrown.
  - A subclass of `Exception` is a checked exception.
- An **unchecked** exception may be caught or re-thrown.
  - A subclass of `RuntimeException` is an unchecked exception.

The checked exception controversy:

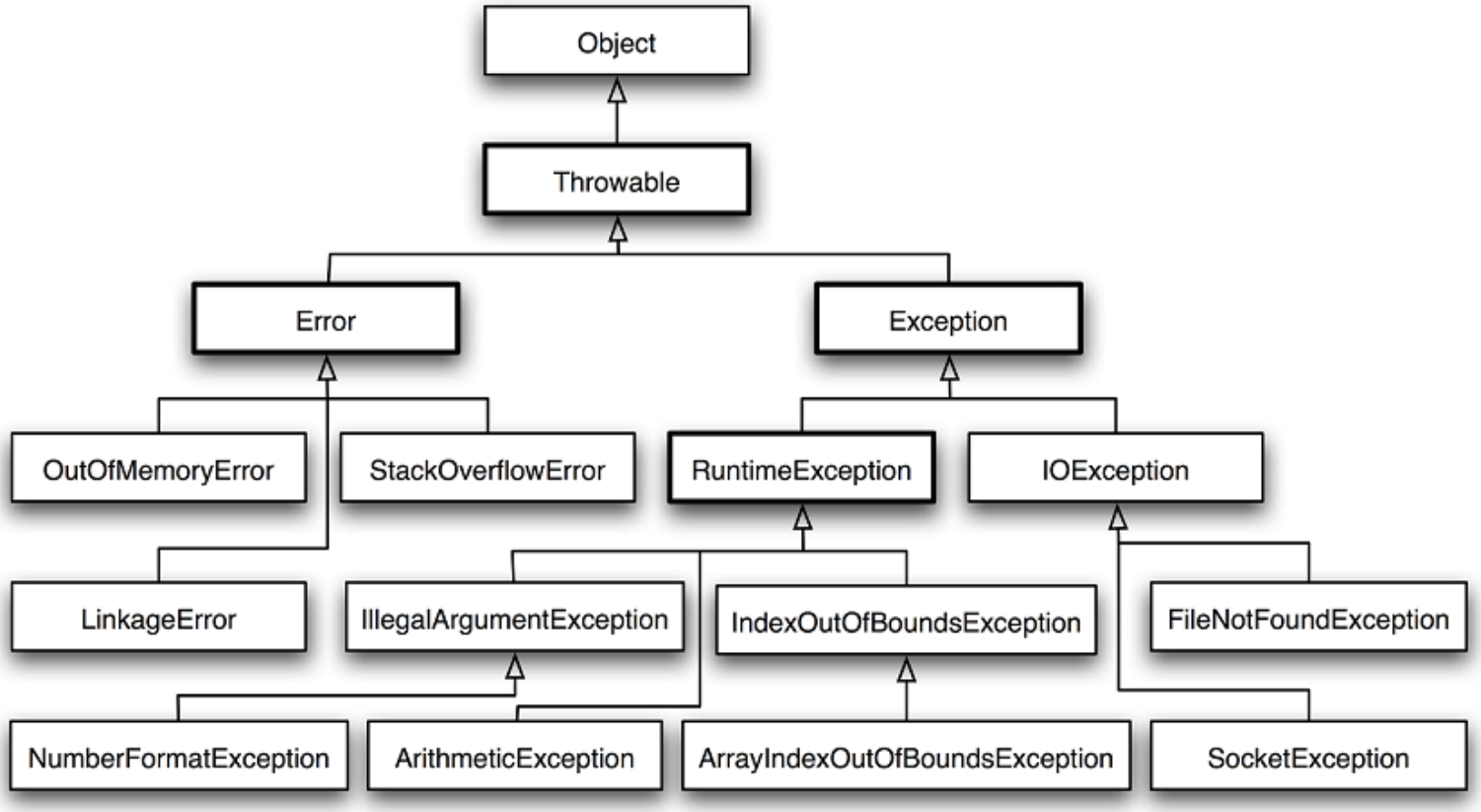
- Programmers dislike being told what to do.
- Painful to use library with lots of checked exceptions.
- Widely considered to be a design mistake.
- Consequently, checked exceptions omitted from C# and C++.

Checked Exceptions

```
public class InsufficientFundsException extends Exception {

    // ...

}
```

```
public void withdraw(int amount) throws InsufficientFundsException {
    // ...

    if (amount >= balance) {
        int deficit = amount - balance;
        throw new InsufficientFundsException(deficit);
    }

    // ...
}
```

Exception Hierarchy



(image source: https://itblackbelt.wordpress.com/2015/02/17/checked-vs-unchecked-exception-in-java-example/)

Enums

Example: Weekdays

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

```
public static void main(String[] args) {
    Day d = Day.FRIDAY;
    switch (r) {
        case MONDAY:
            System.out.println(“Monday! Bummer!”);
        // ...
        case FRIDAY:
            System.out.println(“Aww yeah!”);
        // ...
    }
}
```

Enumerations

We often have data that come in finitely many variants:

- Primary Colors: Red, Green, Blue.
- Operating Systems: Windows, Linux, Mac
- Compass Points: North, East, South, West.
- Deck of Cards: Ace, King, Queen, …
- Weekdays: Monday, Tuesday, Wednesday,…
- Planets: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune.
- …

(Pluto is not a planet)

With Interfaces and Classes

```
public interface Planet {
    String getName();
    double getMass();
}
```

```
public class Mercury extends Planet {
    public String getName() { ... }
    public double getMass() { ... }
}
```

```
public class Venus extends Planet {
    public String getName() { ... }
    public double getMass() { ... }
}
```

Example: Planets

```
enum Planet {
    MERCURY(1),
    VENUS(2),
    EARTH(3),
    MARS(4);

    private int position;

    Planet(int position) {
        this.position = position;
    }

    public int getPosition() {
        return position;
    }
}
```

Example: Planets

```
public static void main(String[] args) {
    Planet p = Planet.EARTH;
    switch (p) {
        case EARTH:
            int pos = p.getPosition();
            System.out.println(“It's earth! Our Homeworld.”);
            System.out.println(“The “ + pos + ” planet.”);
        default:
            System.out.println(“It is not earth.");
    }
}
```

Design Patterns

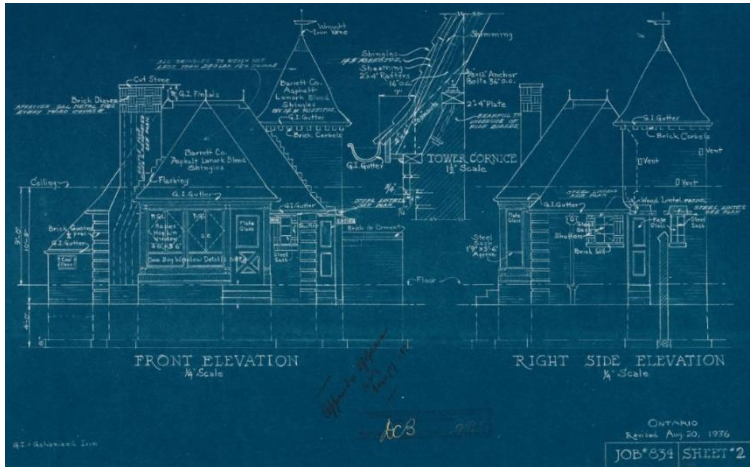
Design Pattern

A blueprint or schematic to solve a re-occurring problem.

- i.e. a common battle-tested solution to a frequently occurring problem.

Each pattern has a **name**, an **intent**, and **solves a specific problem**.

- We can use design patterns to communicate our thoughts!
- Unfortunately this is not always an exact science.



Design Patterns

Creational Patterns:

- Abstract Factory
- **Builder**
- Factory Method
- Singleton

Structural Patterns:

- Adapter
- Composite
- **Decorator**
- Proxy

Behavioral Patterns:

- Command
- Iterator
- Memento
- Null Object
- **Observer**
- State
- **Strategy**
- Visitor

And many many more…

STRATEGY Pattern

Problem:

- We have a class that should vary its behavior depending on circumstance, but the class should not be responsible for the concrete behavior itself.

Solution:

- Abstract the concrete behavior into a STRATEGY interface.
- Augment the class to take an instance of the STRATEGY.
- Implement the varying behavior by delegation to the STRATEGY object.

STRATEGY: Motivation

```
public enum AccountType {
    CHECKING,
    SAVING
}
```

```
public class BankAccount {
    private double balance;
    private AccountType type;
    public BankAccount(double balance, AccountType type) {
        this.balance = balance;
        this.type = type;
    }
    public double getBalance() { ... }
    public void deposit(double amount) { ... }
    public void withdraw(double amount) { ... }
    public void accrueInterest() {
        switch (type) {
            case CHECKING:
                this.balance = this.balance * 1.000001;
                break;
            case SAVING:
                this.balance = this.balance * 1.05;
                break;
        }
    }
}
```

STRATEGY: Interface & Instances

```
public interface InterestStrategy {
    double getInterest(double balance);
}
```

```
public class CheckingAccountInterest implements InterestStrategy {
    public double getInterest(double balance) {
        return balance * 0.000001;
    }
}
```

```
public class SavingsAccountInterest implements InterestStrategy {
    public double getInterest(double balance) {
        return balance * 0.05;
    }
}
```

STRATEGY: Delegate Behavior

```
public class BankAccount {

    private double balance;
    private InterestStrategy interestStrategy;

    public BankAccount(double balance,
                       InterestStrategy interestStrategy) {
        this.balance = balance;
        this.interestStrategy = interestStrategy;
    }

    public double getBalance() { ... }
    public void deposit(double amount) { ... }
    public void withdraw(double amount) { ... }
    public void accrueInterest() {
        this.balance = this.balance +
                       interestStrategy.getInterest(this.balance);
    }
}
```

Observer Pattern

Problem:

- We want to be notified whenever changes happen to a (subject) object.
- For example,
  - Whenever a button is pressed, we want to execute some code.
  - Whenever a key is pressed, we want to execute some code.
  - Whenever a network request completes, we want to execute some code.

Solution:

- We introduce an interface for listeners.
  - i.e. an interface for the methods that should be executed when the event happens.
- We allow clients (other objects) to register listeners on the subject object.
- Whenever an event happens to the subject object, it invokes the listeners.

OBSERVER: Listener Interface

```
public interface AccountListener {

    void notify(double newBalance, double oldBalance);

}
```

OBSERVER: Subject Object

```
public class BankAccount {
    private double balance;
    private List<AccountListener> listeners = new ArrayList();

    public void addListener(AccountListener listener){
        listeners.add(listener);
    }
    public void deposit(double amount) {
        double oldBalance = this.balance;
        this.balance = this.balance + amount;
        for(AccountListener listener: this.listeners){
            listener.notify(this.balance, oldBalance);
        }
    }
}
```

OBSERVER: Listener Object

```
public class BankStatistics implements AccountListener{
    double totalBankBalance;

    public BankStatistics(double totalBankBalance) {
        this.totalBankBalance = totalBankBalance;
    }

    @Override
    public void notify(double newBalance, double oldBalance) {
        double difference = newBalance-oldBalance;
        totalBankBalance+=difference;
    }
}
```

DECORATOR Pattern

Problem:

- We want to augment an existing class with additional behavior.
- For example,
  - To perform logging of some operations performed on the class.
  - To enforce additional security measures.
  - To cache certain computations (e.g. `BufferedReader`)

Solution:

- We refactor the API of the class to an interface.
- We let the original class implement the interface.
- We implement a DECORATOR class which implements the interface.
  - The class contains a reference to an instance of the original class.
  - The class forwards all methods to the original instance.
  - When desired, the class may intercept certain methods and perform additional computation.

DECORATOR: Existing Class

```
public class BankAccount {

    public String getOwner() { ... }

    public double getBalance() { ... }

    public void deposit(double amount) { ... }

    public void withdraw(double amount) { ... }

    public void accrueInterest() { ... }

    public void transferTo(BankAccount otherAccount) { ... }

}
```

DECORATOR: Turn into Interface

```
public interface BankAccount {

    public String getOwner();

    public double getBalance();

    public void deposit(double amount);

    public void withdraw(double amount);

    public void accrueInterest();

    public void transferTo(BankAccount otherAccount);

}
```

DECORATOR: The Decorator

```
public class BankAccountDecorator implements BankAccount {
    private BankAccount account;

    public BankAccountDecorator(BankAccount account) {
        this.account = account;
    }

    public String getOwner() { return account.getOwner(); }
    public double getBalance() { return account.getBalance(); }
    public void deposit(double amount) {
        System.out.println("Deposit: " + amount);
        account.deposit(amount);
    }

    public void transferTo(BankAccount otherAccount) {
        if (account.getOwner().equals(otherAccount.getOwner())) {
            account.transferTo(otherAccount);
        }
    }
}
```

BUILDER Pattern

Problem:

- We want to construct a **complex object**.
- The class of the complex object has many fields.
- Some fields are optional with default values, some are not.
- Some fields have complex invariants, some do not.
- Using a single constructor (or a multitude of constructors) is verbose.

Solution:

- We introduce a **Builder** object to aid construction of the complex object.
- The Builder allows the complex object to be constructed *piece-by-piece*.

BUILDER: The Complex Class

```
public class BankAccount {
    private String firstName;
    private String middleName;
    private String lastName;
    private String socialSecurity;
    private String streetAddress;
    private String streetNumber;
    private String city;
    private String country;
    private double balance;
    private boolean allowOverdraft;
    private double borrowRate;
    private double interestRate;
    private Date accountCreated;
    private Date accountChanged;
    private Date accountArchived;
    private List<Transaction> transactions;
    // ...
}
```

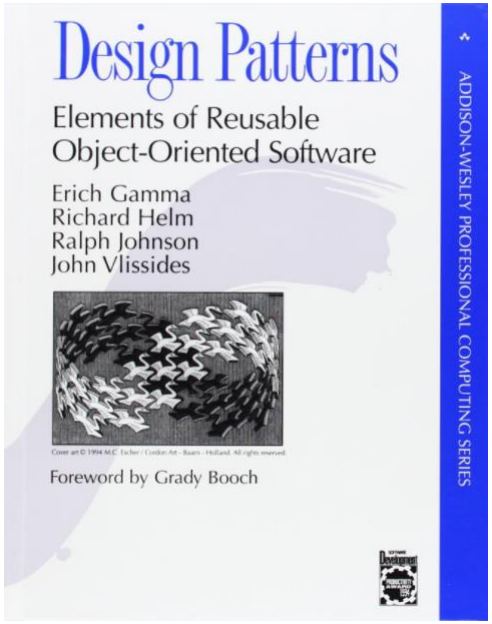
BUILDER: The Builder Class

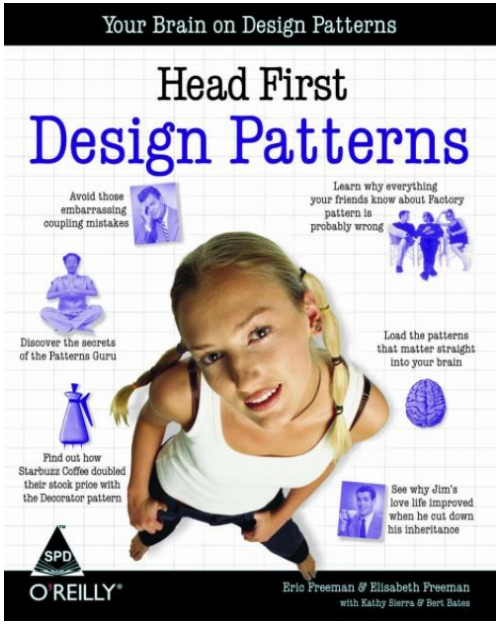
```
public class BankAccountBuilder {
    private String firstName;
    private String middleName;
    private String lastName;
    public BankAccountBuilder setFirstName(String firstName) {
        this.firstName = firstName;
        return this;
    }
    public BankAccountBuilder setMiddleName(String middleName) {
        this.middleName = middleName;
        return this;
    }
    public BankAccountBuilder setLastName(String lastName) {
        this.lastName = lastName;
        return this;
    }
    public BankAccount build() {
        return new BankAccount(...);
    }
}
```

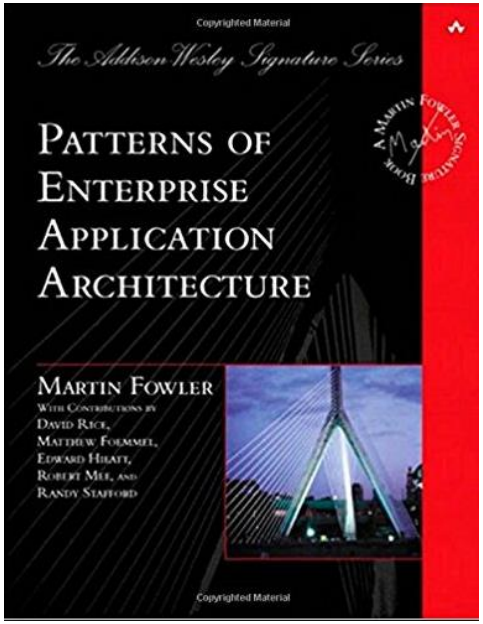
BUILDER: Using It

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

    BankAccountBuilder builder = new BankAccountBuilder();
    builder.setFirstName("Magnus")
           .setLastName("Madsen");
    BankAccount account = builder.build();

}
```

More Design Patterns







Design Patterns II

THE VISITOR PATTERN

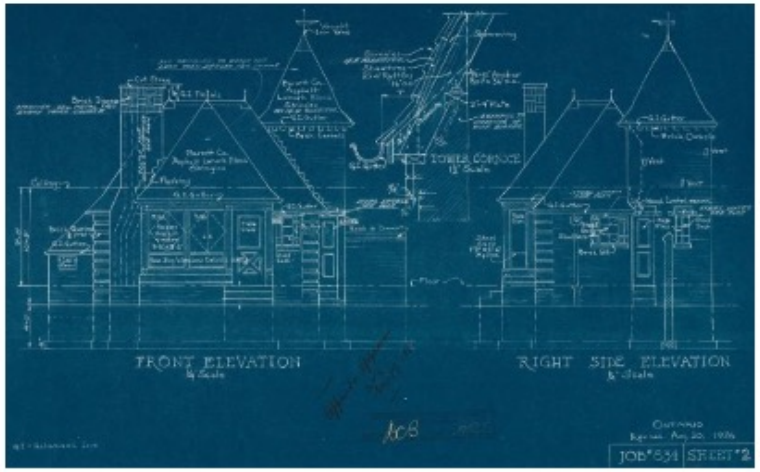
Design Pattern

A blueprint or schematic to solve a re-occurring problem.

- i.e. a common battle-tested solution to a frequently occurring problem.

Each pattern has a **name**, an **intent**, and **solves a specific problem**.

- We can use design patterns to communicate our thoughts!
- Unfortunately this is not always an exact science.



Design Patterns

Creational Patterns:

- Abstract Factory
- **Builder**
- Factory Method
- Singleton

Structural Patterns:

- Adapter
- Composite
- **Decorator**
- Proxy

Behavioral Patterns:

- Command
- Iterator
- Memento
- Null Object
- **Observer**
- State
- **Strategy**
- Visitor

And many many more…

The Visitor Pattern

- **Purpose**: Define new functionalities without introducing the modifications to an existing class.

- Usually we start from a **composite class**:
  - List, Trees, Graphs, etc. of different elemts.

- Implement **new functionalites** by transvering elements.
  - No coded can be added to your composite class.

- **Visitor Pattern:**
  - Implement different ways of visiting the elements of the composite class
  - No code added to your composite class.

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.