Čo nás čaká a neminie...

1. časť

- Úvod do Javy
- Štruktúra Platformy a EA
- Vývojové Technológie (BIZ)
- Kolekcie (APP)
- Logovanie
- Lokalizácia

2. časť

- JDBC a DBMS (TECH)
- XML, NIO2
- Regulárne Výrazy
- Prehľad EA
- Támec TOGAF a ADM
- Prehľad JEE a .NET

Best practices | Faily

Čo nás čaká a neminie...

1. časť

Architektúra

2. časť

Java

Best practices | Faily

Prečo by mal každý programátor ovládať kolekcie?

slido

Join at

slido.com

#VAVA4

1: Poll

Show results

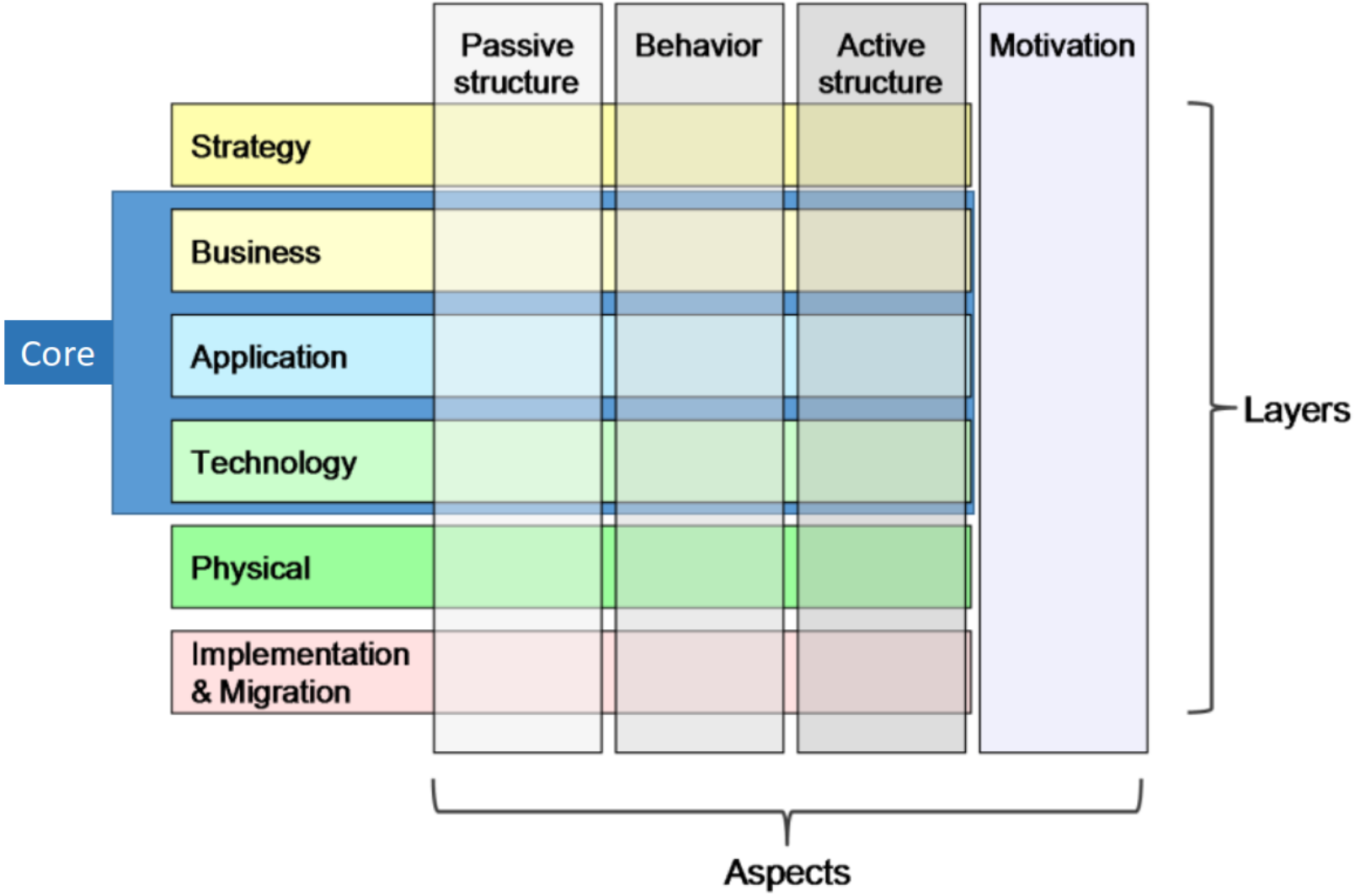
Showing Q&A

SLIDO – VAVA4

1. Ktoré Java kolekcie sú najčastejšie používané vo vývoji aplikácií a prečo?
2. Z hľadiska Enterprise Architektúry a ArchiMate, je kolekcia aký typ elementu a prečo?
3. Čo je kolekcia z pohľadu Javy?

https://wall.sli.do/event/sW3ieupQnVBySBmFiw67rk?section=d1135817-a1b7-4ba2-a55a-726aa59e59b2

Úplný rámec - vrstvy a aspekty



Aspekty (Aspects)

ArchiMate striktne oddeľuje 3 základné aspekty:

1. **Active Structure Aspect** – reprezentujú **štrukturálne elementy** napr. aktérov, role, komponenty, hardvér, ktoré sú schopné určitého chovania. Tiež sú tieto elementy označované ako aktívne.
2. **Behavior Aspect** – Reprezentuje **chovanie** (napr. procesy, funkcie, služby) **vykonávané aktívnymi** (štrukturálnymi) **elementami**. Chovanie je priradené (assigned) aktívnemu elementu.
3. **Passive Structure Aspect** (pasívne elementy) – **reprezentujú objekty** (napr. biznis objekty, dátové objekty ale aj fyzické objekty), na ktorých môže byť aplikované chovanie aktívneho elementu (či sú takýmto chovaním ovplyvnený)

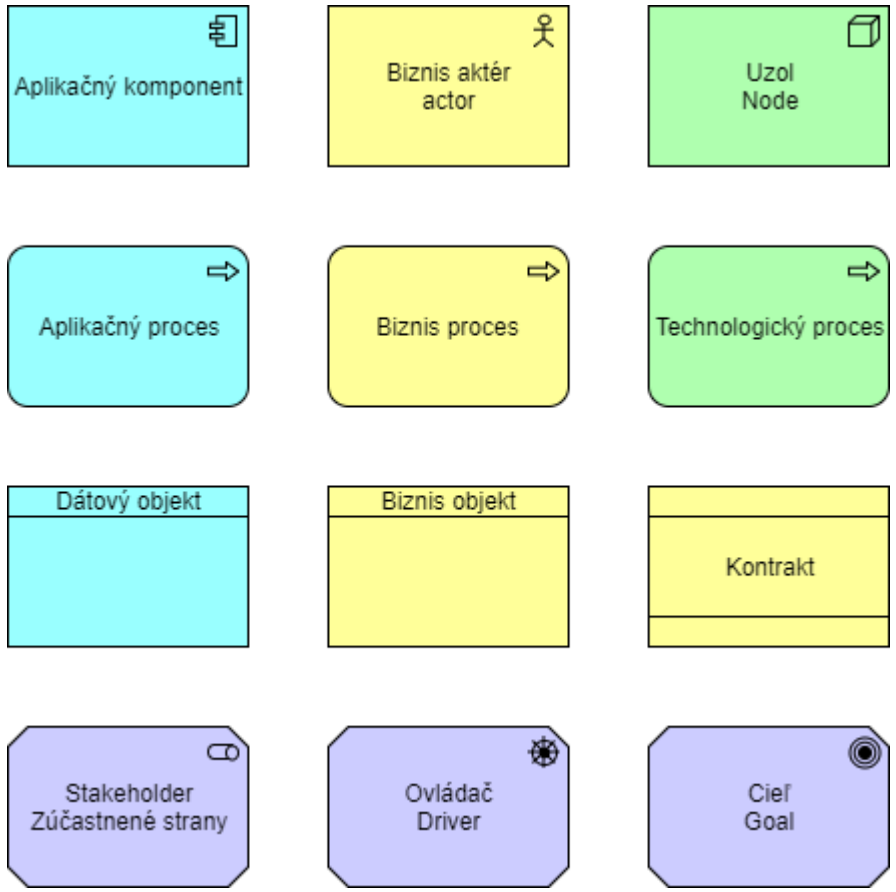
Pochopenie typov aspektov elementu

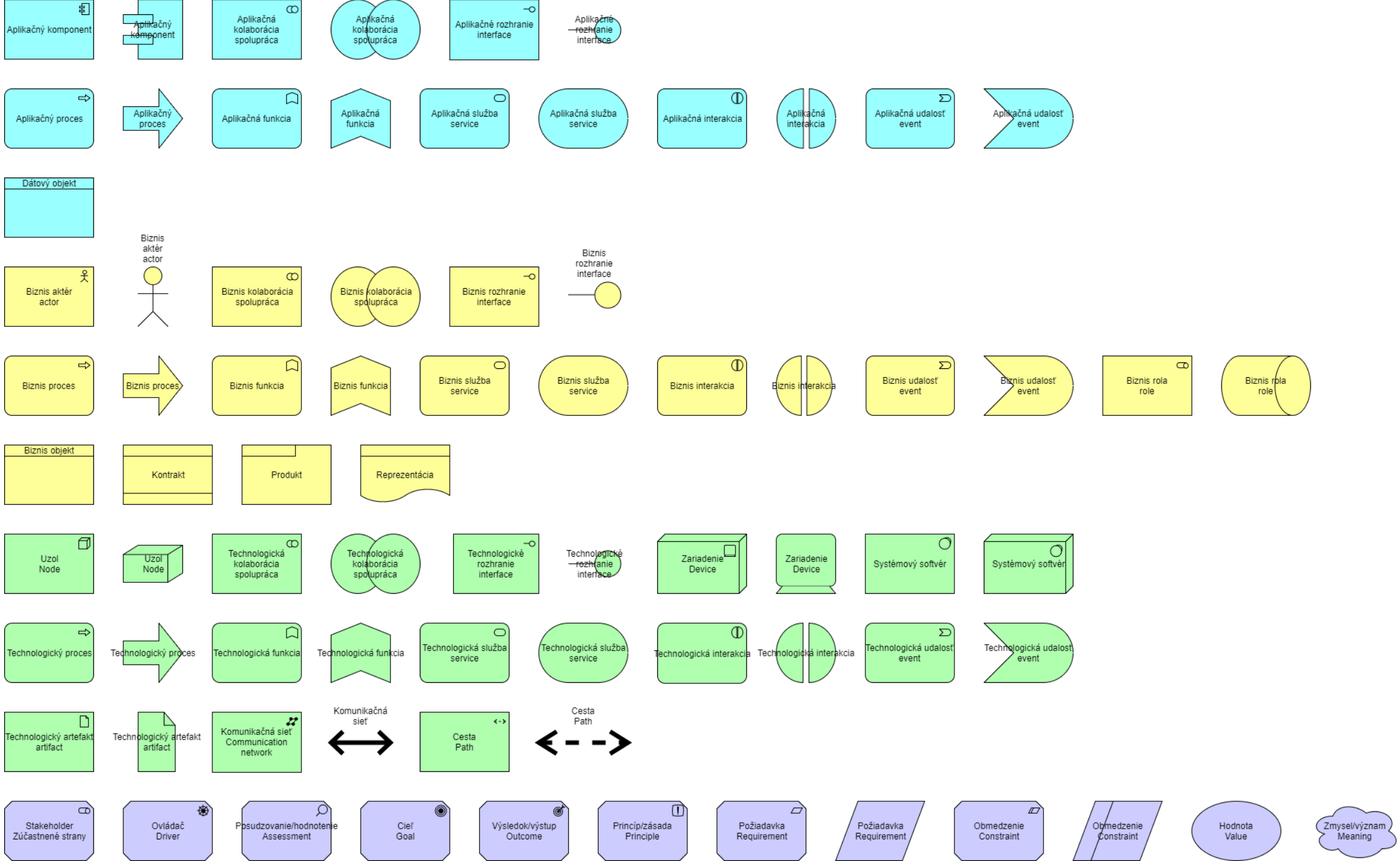
- Aká je aktívna štruktúra, behaviorálna a pasívna štruktúra?
  - Kuchár varí jedlo
  - Mechanik zvára auto
- **Kuchár/Mechanik**
  - **Aktívny prvok** štruktúry schopný **vykonávať správanie, zobrazuje** správanie – subject/**podmet** (**podstatné meno**)
- **Varenie/zváranie**
  - **Behaviorálny** prvok je jednotka **aktivity vykonávaná aktívnym elementom** (štruktúrami) - typicky verb/**prísudok** (**sloveso**)
- **Potraviny/Auto**
  - **Prvok pasívnej** štruktúry, **na ktorom sa vykonáva správanie** – objekt/**predmet** (**podstatné meno**) (informácie, údaje alebo fyzické)

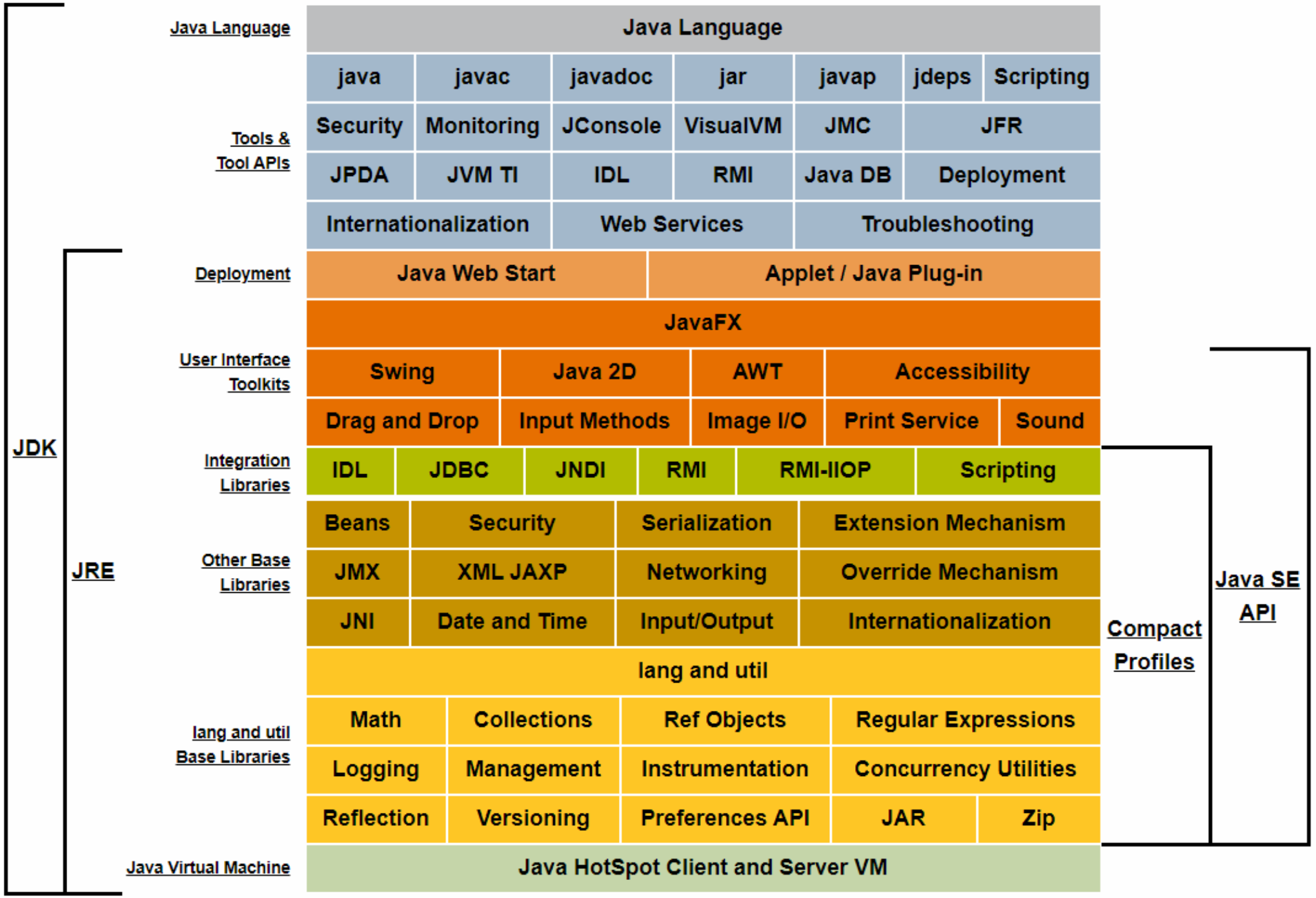
Inšpirácia prirodzeným jazykom

Rozlíšenie elementov podľa rohov

1. Ostré rohy – aktívne elementy
2. Zaoblené rohy – element chovania (behavior)
3. Ostrý roh s preškrtnutím – pasívne elementy
4. Skosené rohy – motivačný element

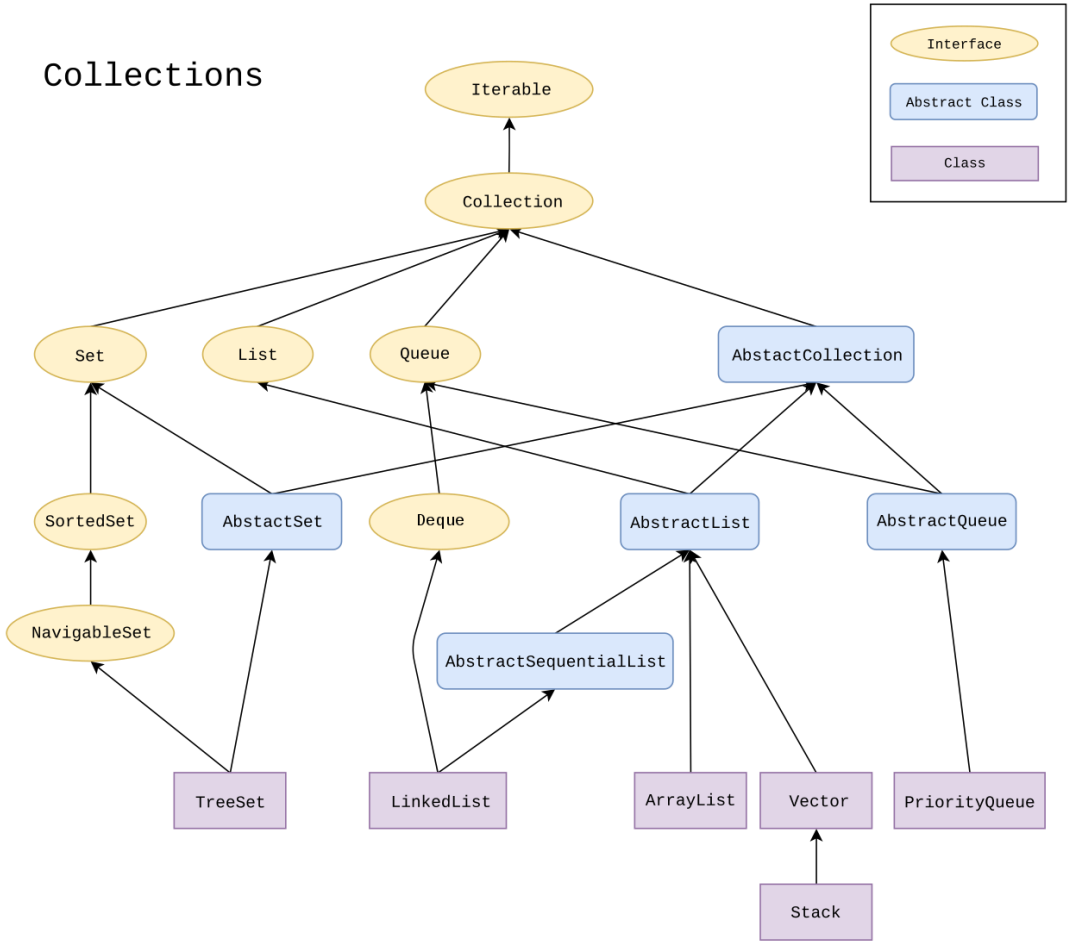



Aplikačný komponent
Aplikačný komponent
Aplikačná kolaborácia spolupráca
Aplikačná kolaborácia spolupráca
Aplikačné rozhranie interface
Aplikačné rozhranie interface
Aplikačný proces
Aplikačný proces
Aplikačná funkcia
Aplikačná funkcia
Aplikačná služba service
Aplikačná služba service
Aplikačná interakcia
Aplikačná interakcia
Aplikačná udalosť event
Aplikačná udalosť event
Dátový objekt
Biznis aktér actor
Biznis aktér actor
Biznis kolaborácia spolupráca
Biznis kolaborácia spolupráca
Biznis rozhranie interface
Biznis rozhranie interface
Biznis proces
Biznis proces
Biznis funkcia
Biznis funkcia
Biznis služba service
Biznis služba service
Biznis interakcia
Biznis interakcia
Biznis udalosť event
Biznis udalosť event
Biznis rola role
Biznis rola role
Biznis objekt
Kontrakt
Produkt
Reprezentácia
Uzol Node
Uzol Node
Technologická kolaborácia spolupráca
Technologická kolaborácia spolupráca
Technologické rozhranie interface
Technologické rozhranie interface
Zariadenie Device
Zariadenie Device
Systémový softvér
Systémový softvér
Technologický proces
Technologický proces
Technologická funkcia
Technologická funkcia
Technologická služba service
Technologická služba service
Technologická interakcia
Technologická interakcia
Technologická udalosť event
Technologická udalosť event
Technologický artefakt artifact
Technologický artefakt artifact
Komunikačná sieť Communication network
Komunikačná sieť
Cesta Path
Cesta Path
Stakeholder Zúčastnené strany
Ovládač Driver
Posudzovanie/hodnotenie Assessment
Cieľ Goal
Výsledok/výstup Outcome
Princíp/zásada Principle
Požiadavka Requirement
Požiadavka Requirement
Obmedzenie Constraint
Obmedzenie Constraint
Hodnota Value
Zmysel/význam Meaning


JDK
JRE
Java Language
Java Language
java
javac
javadoc
jar
javap
jdeps
Scripting
Tools &
Tool APIs
Security
Monitoring
JConsole
VisualVM
JMC
JFR
JPDA
JVM TI
IDL
RMI
Java DB
Deployment
Internationalization
Web Services
Troubleshooting
Deployment
Java Web Start
Applet / Java Plug-in
JavaFX
User Interface
Toolkits
Swing
Java 2D
AWT
Accessibility
Drag and Drop
Input Methods
Image I/O
Print Service
Sound
Integration
Libraries
IDL
JDBC
JNDI
RMI
RMI-IIOP
Scripting
Other Base
Libraries
Beans
Security
Serialization
Extension Mechanism
JMX
XML JAXP
Networking
Override Mechanism
JNI
Date and Time
Input/Output
Internationalization
lang and util
lang and util
Base Libraries
Math
Collections
Ref Objects
Regular Expressions
Logging
Management
Instrumentation
Concurrency Utilities
Reflection
Versioning
Preferences API
JAR
Zip
Java Virtual Machine
Java HotSpot Client and Server VM
Compact
Profiles
Java SE
API

Java collections framework (JCF)

- Je **súbor tried a rozhraní**, ktoré implementujú bežne **znovu použiteľné kolekcie dátových štruktúr**
- Aj keď sa označuje ako rámec, **funguje na spôsob knižnice**
- Rámec kolekcií poskytuje **rozhrania**, ktoré definujú rôzne kolekcie, aj **triedy**, ktoré **ich implementujú**



Od JDK 1.2

Príklady kolekcií

1. Čakatelia na pozíciu profesora
2. Ašpiranti na nobelové ceny
3. Zoznam študentov s ID
4. Hra poker (kolekcia kariet)
5. Poštová schránka s listami (kolekcia listov)
6. Telefónny zoznam (priradenie mien k telefónnym číslam)
7. Zoznam tvarov vo vzorkovnici modelovacieho nástroja

Architektúra kolekcií

- **Takmer všetky** kolekcie v Jave sú **odvodené z rozhrania java.util.Collection**
- Rozhranie **Collection** definuje **základné časti všetkých kolekcií**
- Rozhranie uvádza **metódy add()** a **remove()** na pridávanie a odstraňovanie z kolekcie
- Vyžaduje sa aj metóda **toArray(),** ktorá **konvertuje** kolekciu **na** jednoduché **pole** všetkých prvkov v kolekcii
- Metóda **contains()** skontroluje, či je **zadaný prvok v kolekcii**
- Rozhranie Collection je podrozhraním java.lang.**Iterable**, takže každá kolekcia môže používať **for-each**
- Rozhranie Iterable poskytuje metódu iterator() používanú príkazmi for-each
- Všetky kolekcie majú **iterator**, ktorý **prechádza všetkými prvkami v kolekcii**
- Kolekcia je **generická**
- Akákoľvek kolekcia môže byť napísaná na uloženie akejkoľvek triedy

Ordered lists (Zoznamy)

Maps (Dictionaries)

Sets (Množiny)

The Java™ Tutorials

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

The Java Tutorials are practical guides for programmers who want to use the Java programming language to create applications. They include hundreds of complete, working examples, and dozens of lessons. Groups of related lessons are organized into "trails".

Trails Covering the Basics

These trails are available in book form as *The Java Tutorial, Sixth Edition*. To buy this book, refer to the box to the right.

- Getting Started — An introduction to Java technology and lessons on installing Java development software and using it to create a simple program.
- Learning the Java Language — Lessons describing the essential concepts and features of the Java Programming Language.
- Essential Java Classes — Lessons on exceptions, basic input/output, concurrency, regular expressions, and the platform environment.
- Collections — Lessons on using and extending the Java Collections Framework.
- Date-Time APIs — How to use the `java.time` pages to write date and time code.
- Deployment — How to package applications and applets using JAR files, and deploy them using Java Web Start and Java Plug-in.
- Preparation for Java Programming Language Certification — List of available training and tutorial resources.

Creating Graphical User Interfaces

- Creating a GUI with Swing — A comprehensive introduction to GUI creation on the Java platform.
- Creating a JavaFX GUI — A collection of JavaFX tutorials.

Specialized Trails and Lessons

These trails and lessons are only available as web pages.

- Custom Networking — An introduction to the Java platform's powerful networking features.
- The Extension Mechanism — How to make custom APIs available to all applications running on the Java platform.
- Full-Screen Exclusive Mode API — How to write applications that more fully utilize the user's graphics hardware.
- Generics — An enhancement to the type system that supports operations on objects of various types while providing compile-time type safety. Note that this lesson is for advanced users. The Java Language trail contains a Generics lesson that is suitable for beginners.
- Internationalization — An introduction to designing software so that it can be easily adapted (localized) to various languages and regions.
- JavaBeans — The Java platform's component technology.
- JAXB — Introduces the Java architecture for XML Binding (JAXB) technology.
- JAXP — Introduces the Java API for XML Processing (JAXP) technology.

Not sure where to start?
See Learning Paths

Tutorial Contents

Really Big Index

Tutorial Resources

- Last Updated 2022-03-04
- The Java Tutorials' Blog has news and updates about the Java SE tutorials.
- Download the latest Java Tutorials bundle.

In Book Form

- *The Java Tutorial, Sixth Edition.* Amazon.com.

Other Resources

- Java SE 8 Developer Guides
- JDK 8 API Documentation

Oracle Training and

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Trail: Collections

This section describes the Java Collections Framework. Here, you will learn what collections are and how they can make your job easier and programs better. You'll learn about the core elements — interfaces, implementations, aggregate operations, and algorithms — that comprise the Java Collections Framework.

Introduction tells you what collections are, and how they'll make your job easier and your programs better. You'll learn about the core elements that comprise the Collections Framework: *interfaces*, *implementations* and *algorithms*.

Interfaces describes the *core collection interfaces*, which are the heart and soul of the Java Collections Framework. You'll learn general guidelines for effective use of these interfaces, including when to use which interface. You'll also learn idioms for each interface that will help you get the most out of the interfaces.

Aggregate Operations iterate over collections on your behalf, which enable you to write more concise and efficient code that process elements stored in collections.

Implementations describes the JDK's *general-purpose collection implementations* and tells you when to use which implementation. You'll also learn about the *wrapper implementations*, which add functionality to general-purpose implementations.

Algorithms describes the *polymorphic algorithms* provided by the JDK to operate on collections. With any luck you'll never have to write your own sort routine again!

Custom Implementations tells you why you might want to write your own collection implementation (instead of using one of the general-purpose implementations provided by the JDK), and how you'd go about it. It's easy with the JDK's *abstract collection implementations*!

Interoperability tells you how the Collections Framework interoperates with older APIs that predate the addition of Collections to Java. Also, it tells you how to design new APIs so that they'll interoperate seamlessly with other new APIs.

Lesson: Introduction to Collections

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

What Is a Collections Framework?

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

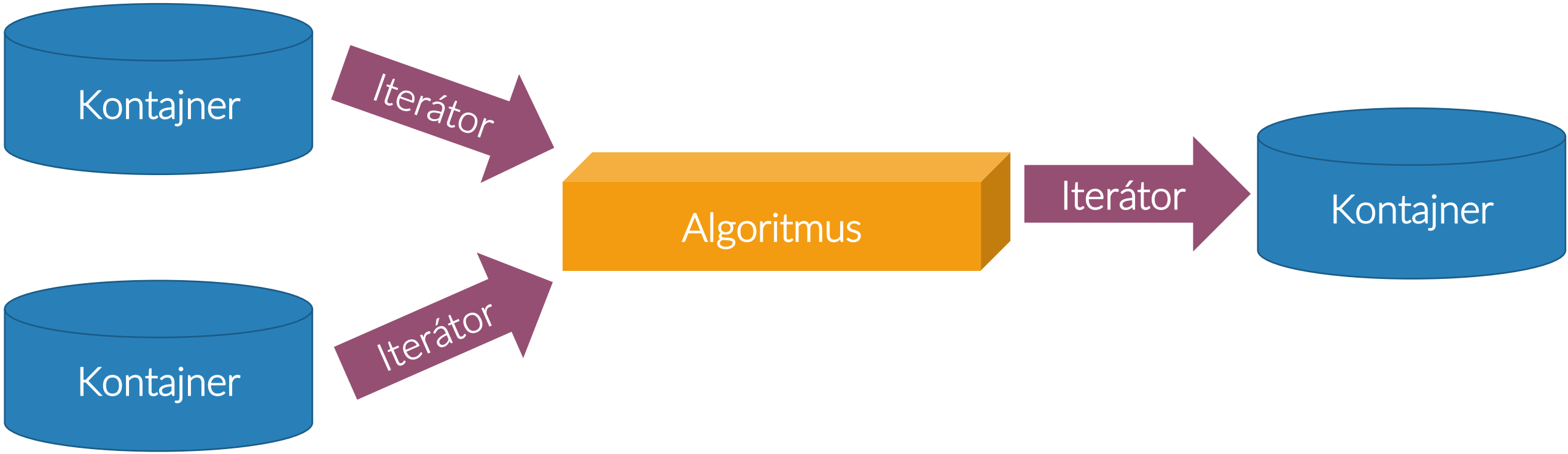
- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

Benefits of the Java Collections Framework

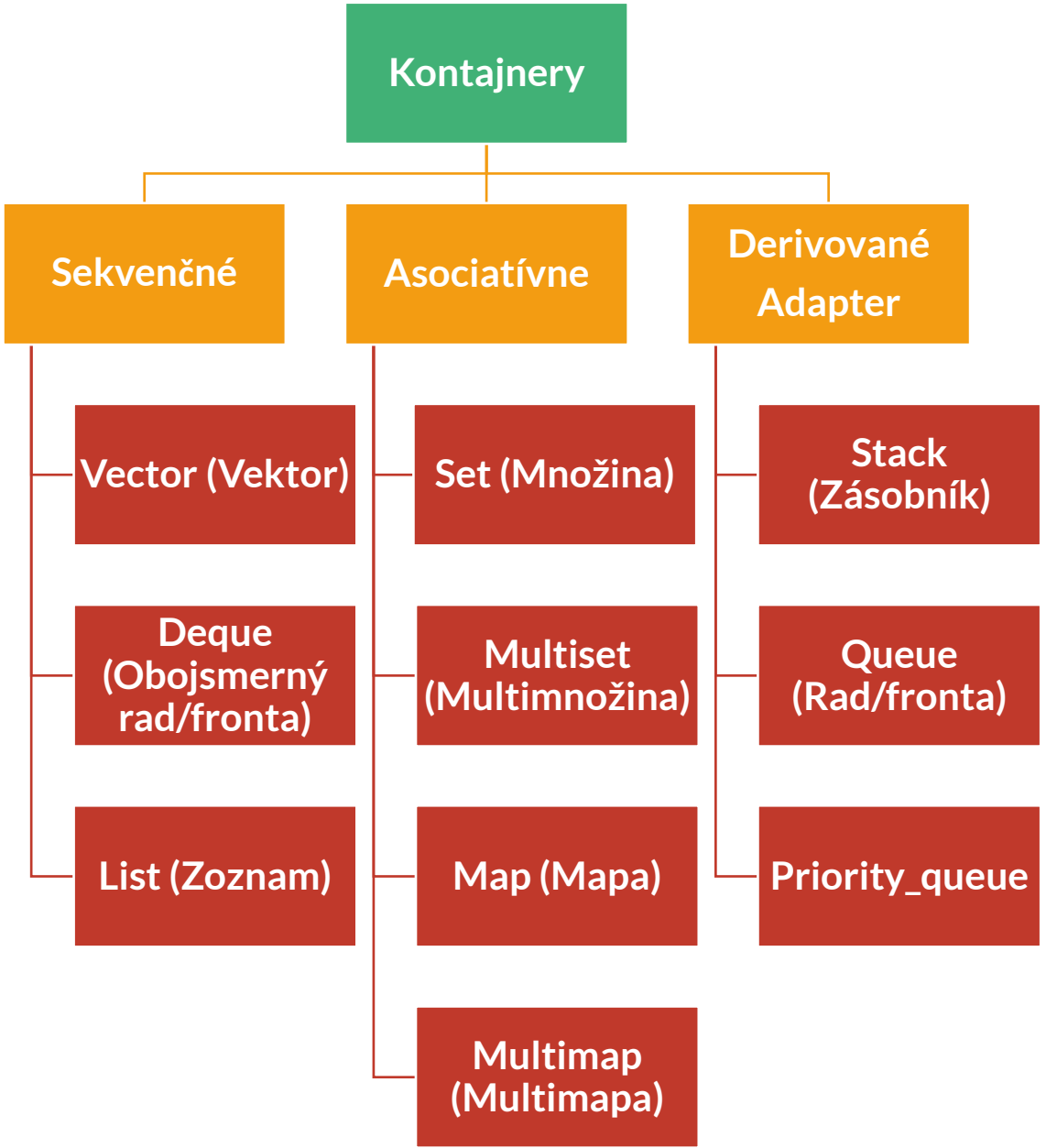
The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.


Kontajner
Iterátor
Kontajner
Iterátor
Algoritmus
Iterátor
Kontajner

Kontajnery

- **Sekvenčné kontajnery** umožňujú **pracovať** s **prvkami**, ktoré sú **uložené v určitom poradí** (**sekvencii**)
  - **Pristupujeme** k nim **cez indexy**, prípadne pomocou **iterátorov**.
  - Veľkou výhodou sekvenčných kontajnerov je **možnosť pracovať** s **údajmi dynamicky**, t.j. zvyšovať alebo znižovať ich počet
- Okrem sekvenčných kontajnerov extistuje aj ďalšia skupina kontajnerov, tzv. **asociatívne** a **derivované kontajnery**



Príklady štandardných knižníc

C	C++	Python	Go	Java
• assert • float • math • stdio • stdlib • string • threads	• list • map • deque • vector • algorithm • iterator • string • regex • fstream • iostream • exception	• list • dict • tuple • range • open • enum • collections • math • pickle • zlib • threading • queue • email • json	• tar • gzip • heap • crypto • hash • image • io • mime • net • os • path • strings • time • unicode	• java.lang • java.io java.net • java.math • java.text • java.awt.image • java.security • java.beans

Module java.base
Package java.util

Class Collections

java.lang.Object
    java.util.Collections

```
public class Collections
extends Object
```

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are null.

The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort` does not have to be a mergesort, but it does have to be *stable*.)

The "destructive" algorithms contained in this class, that is, the algorithms that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if the collection does not support the appropriate mutation primitive(s), such as the `set` method. These algorithms may, but are not required to, throw this exception if an invocation would have no effect on the collection. For example, invoking the `sort` method on an unmodifiable list that is already sorted may or may not throw `UnsupportedOperationException`.

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `Set`, `List`, `Map`

Field Summary

Module java.base
Package java.util

Interface Collection<E>

Type Parameters:

E - the type of elements in this collection

All Superinterfaces:

Iterable<E>

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, EventSet, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentHashMap.KeySetView, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

---

```
public interface Collection<E>
extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.

All general-purpose `Collection` implementation classes (which typically implement `Collection` indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type `Collection`, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose `Collection` implementations in the Java platform libraries

Module java.base
Package java.util

Interface Iterator<E>

Type Parameters:

E - the type of elements returned by this iterator

All Known Subinterfaces:

EventIterator, ListIterator<E>, PrimitiveIterator<T,T_CONS>, PrimitiveIterator.OfDouble, PrimitiveIterator.OfInt, PrimitiveIterator.OfLong, XMLEventReader

All Known Implementing Classes:

BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner

```
public interface Iterator<E>
```

An iterator over a collection. `Iterator` takes the place of `Enumeration` in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

API Note:

An `Enumeration` can be converted into an `Iterator` by using the `Enumeration.asIterator()` method.

Since:

1.2

See Also:

`Collection`, `ListIterator`, `Iterable`

Method Summary

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Lesson: Implementations

Implementations are the data objects used to store collections, which implement the interfaces described in the Interfaces section. This lesson describes the following kinds of implementations:

- **General-purpose implementations** are the most commonly used implementations, designed for everyday use. They are summarized in the table titled General-purpose-implementations.
- **Special-purpose implementations** are designed for use in special situations and display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations** are designed to support high concurrency, typically at the expense of single-threaded performance. These implementations are part of the `java.util.concurrent` package.
- **Wrapper implementations** are used in combination with other types of implementations, often the general-purpose ones, to provide added or restricted functionality.
- **Convenience implementations** are mini-implementations, typically made available via static factory methods, that provide convenient, efficient alternatives to general-purpose implementations for special collections (for example, singleton sets).
- **Abstract implementations** are skeletal implementations that facilitate the construction of custom implementations — described later in the Custom Collection Implementations section. An advanced topic, it's not particularly difficult, but relatively few people will need to do it.

The general-purpose implementations are summarized in the following table.

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

As you can see from the table, the Java Collections Framework provides several general-purpose implementations of the `Set`, `List` , and `Map` interfaces. In each case, one implementation — `HashSet`, `ArrayList`, and `HashMap` — is clearly the one to use for most applications, all other things being equal. Note that the `SortedSet` and the `SortedMap` interfaces do not have rows in the table. Each of those interfaces has one implementation (`TreeSet` and `TreeMap`) and is listed in the `Set` and the `Map` rows. There are two general-purpose `Queue` implementations — `LinkedList`, which is also a `List` implementation, and `PriorityQueue`, which is omitted from the table. These two implementations provide very different semantics: `LinkedList` provides

JDK Release Notes

Oracle Java SE Publicly Available Release Families

- JDK 17 Release notes
- JDK 16 Release notes
- JDK 15 Release notes
- JDK 14 Release notes
- JDK 13 Release notes
- JDK 12 Release notes
- JDK 11 Release notes
- JDK 10 Release notes
- JDK 9 Release Notes
- JDK 8 Release Notes
- JDK 7 Release Notes for update releases
- JDK 7 Release Notes for revision builds
- JDK 6 Release Notes post 6u45, and revision builds of earlier releases
- JDK 6 Release Notes through 6u45

Oracle Java SE Commercial Offering Releases

JDK 7 and JDK 8 release families that have reached end-of-public-updates but are still supported for Oracle Customers. To access the latest releases of these families you will need to license Java SE Advanced.

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Set Implementations

The `Set` implementations are grouped into general-purpose and special-purpose implementations.

General-Purpose Set Implementations

There are three general-purpose `Set` implementations — `HashSet`, `TreeSet`, and `LinkedHashSet`. Which of these three to use is generally straightforward. `HashSet` is much faster than `TreeSet` (constant-time versus log-time for most operations) but offers no ordering guarantees. If you need to use the operations in the `SortedSet` interface, or if value-ordered iteration is required, use `TreeSet`; otherwise, use `HashSet`. It's a fair bet that you'll end up using `HashSet` most of the time.

`LinkedHashSet` is in some sense intermediate between `HashSet` and `TreeSet`. Implemented as a hash table with a linked list running through it, it provides *insertion-ordered* iteration (least recently inserted to most recently) and runs nearly as fast as `HashSet`. The `LinkedHashSet` implementation spares its clients from the unspecified, generally chaotic ordering provided by `HashSet` without incurring the increased cost associated with `TreeSet`.

One thing worth keeping in mind about `HashSet` is that iteration is linear in the sum of the number of entries and the number of buckets (the *capacity*). Thus, choosing an initial capacity that's too high can waste both space and time. On the other hand, choosing an initial capacity that's too low wastes time by copying the data structure each time it's forced to increase its capacity. If you don't specify an initial capacity, the default is 16. In the past, there was some advantage to choosing a prime number as the initial capacity. This is no longer true. Internally, the capacity is always rounded up to a power of two. The initial capacity is specified by using the `int` constructor. The following line of code allocates a `HashSet` whose initial capacity is 64.

```
Set<String> s = new HashSet<String>(64);
```

The `HashSet` class has one other tuning parameter called the *load factor*. If you care a lot about the space consumption of your `HashSet`, read the `HashSet` documentation for more information. Otherwise, just accept the default; it's almost always the right thing to do.

If you accept the default load factor but want to specify an initial capacity, pick a number that's about twice the size to which you expect the set to grow. If your guess is way off, you may waste a bit of space, time, or both, but it's unlikely to be a big problem.

`LinkedHashSet` has the same tuning parameters as `HashSet`, but iteration time is not affected by capacity. `TreeSet` has no tuning parameters.

Special-Purpose Set Implementations

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

List Implementations

`List` implementations are grouped into general-purpose and special-purpose implementations.

General-Purpose List Implementations

There are two general-purpose `List` implementations — `ArrayList` and `LinkedList`. Most of the time, you'll probably use `ArrayList`, which offers constant-time positional access and is just plain fast. It does not have to allocate a node object for each element in the `List`, and it can take advantage of `System.arraycopy` when it has to move multiple elements at the same time. Think of `ArrayList` as `Vector` without the synchronization overhead.

If you frequently add elements to the beginning of the `List` or iterate over the `List` to delete elements from its interior, you should consider using `LinkedList`. These operations require constant-time in a `LinkedList` and linear-time in an `ArrayList`. But you pay a big price in performance. Positional access requires linear-time in a `LinkedList` and constant-time in an `ArrayList`. Furthermore, the constant factor for `LinkedList` is much worse. If you think you want to use a `LinkedList`, measure the performance of your application with both `LinkedList` and `ArrayList` before making your choice; `ArrayList` is usually faster.

`ArrayList` has one tuning parameter — the *initial capacity*, which refers to the number of elements the `ArrayList` can hold before it has to grow. `LinkedList` has no tuning parameters and seven optional operations, one of which is `clone`. The other six are `addFirst`, `getFirst`, `removeFirst`, `addLast`, `getLast`, and `removeLast`. `LinkedList` also implements the `Queue` interface.

Special-Purpose List Implementations

`CopyOnWriteArrayList` is a `List` implementation backed up by a copy-on-write array. This implementation is similar in nature to `CopyOnWriteArraySet`. No synchronization is necessary, even during iteration, and iterators are guaranteed never to throw `ConcurrentModificationException`. This implementation is well suited to maintaining event-handler lists, in which change is infrequent, and traversal is frequent and potentially time-consuming.

If you need synchronization, a `Vector` will be slightly faster than an `ArrayList` synchronized with `Collections.synchronizedList`. But `Vector` has loads of legacy operations, so be careful to always manipulate the `Vector` with the `List` interface or else you won't be able to replace the implementation at a later time.

If your `List` is fixed in size — that is, you'll never use `remove`, `add`, or any of the bulk operations other than `containsAll` — you have a third option that's definitely worth considering. See `Arrays.asList` in the Convenience Implementations section for more information.

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Map Implementations

`Map` implementations are grouped into general-purpose, special-purpose, and concurrent implementations.

General-Purpose Map Implementations

The three general-purpose `Map` implementations are `HashMap`, `TreeMap` and `LinkedHashMap`. If you need `SortedMap` operations or key-ordered `Collection`-view iteration, use `TreeMap`; if you want maximum speed and don't care about iteration order, use `HashMap`; if you want near-`HashMap` performance and insertion-order iteration, use `LinkedHashMap`. In this respect, the situation for `Map` is analogous to `Set`. Likewise, everything else in the Set Implementations section also applies to `Map` implementations.

`LinkedHashMap` provides two capabilities that are not available with `LinkedHashSet`. When you create a `LinkedHashMap`, you can order it based on key access rather than insertion. In other words, merely looking up the value associated with a key brings that key to the end of the map. Also, `LinkedHashMap` provides the `removeEldestEntry` method, which may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map. This makes it very easy to implement a custom cache.

For example, this override will allow the map to grow up to as many as 100 entries and then it will delete the eldest entry each time a new entry is added, maintaining a steady state of 100 entries.

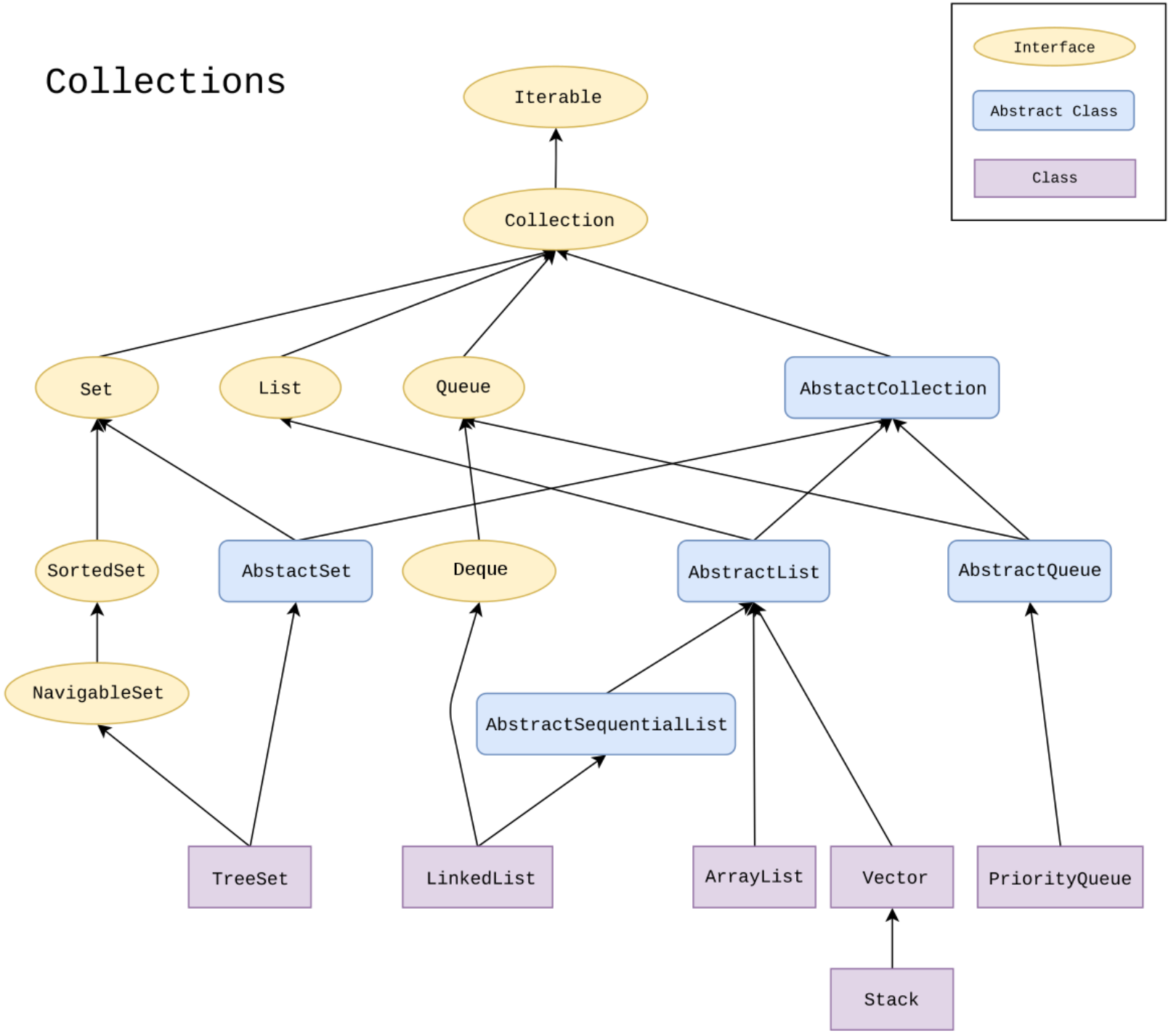
```
private static final int MAX_ENTRIES = 100;

protected boolean removeEldestEntry(Map.Entry eldest) {
    return size() > MAX_ENTRIES;
}
```

Special-Purpose Map Implementations

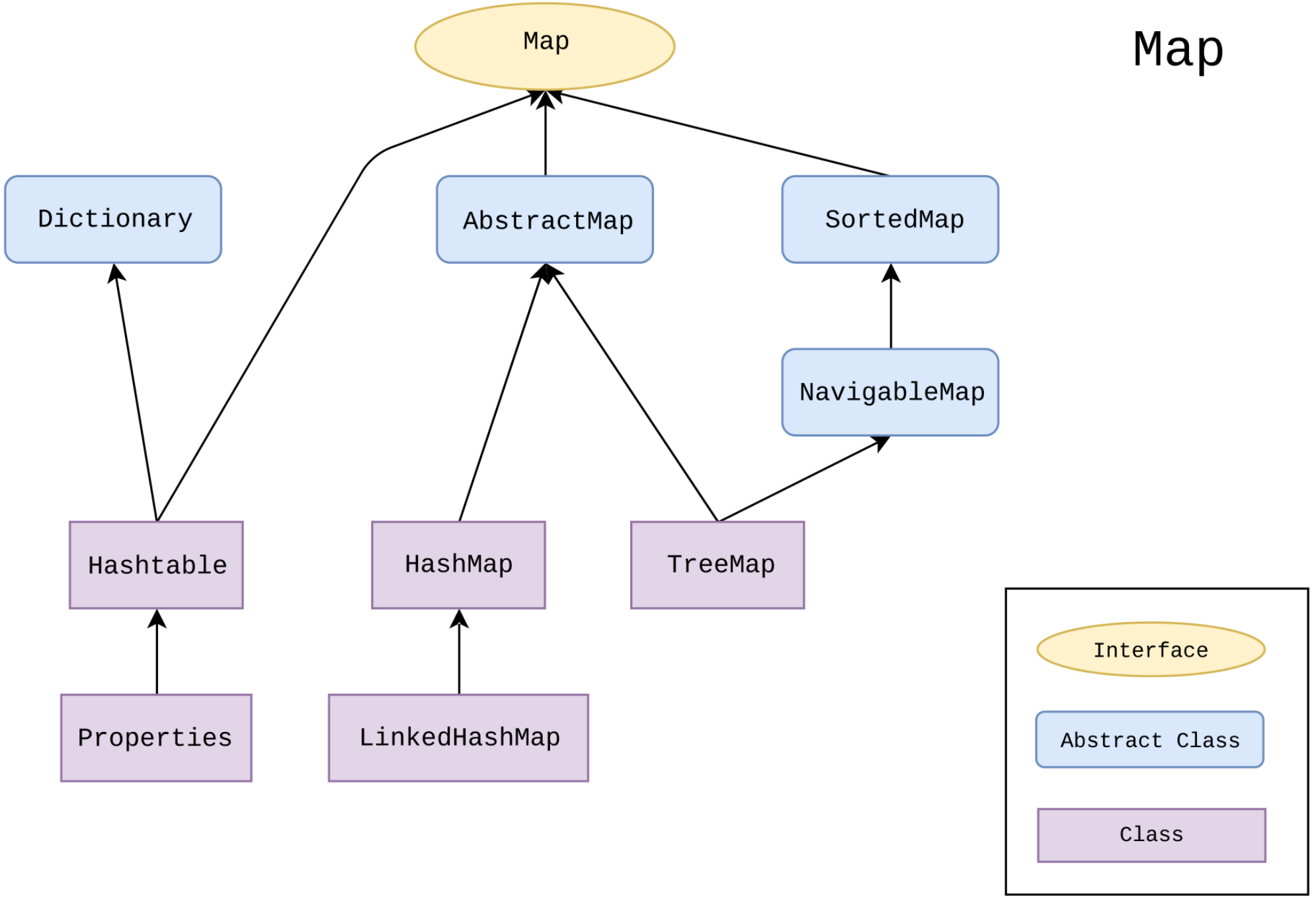
There are three special-purpose Map implementations — `EnumMap`, `WeakHashMap` and `IdentityHashMap`. `EnumMap`, which is internally implemented as an `array`, is a high-performance `Map` implementation for use with enum keys. This implementation combines the richness and safety of the `Map` interface with a speed approaching that of an array. If you want to map an enum to a value, you should always use an `EnumMap` in preference to an array.

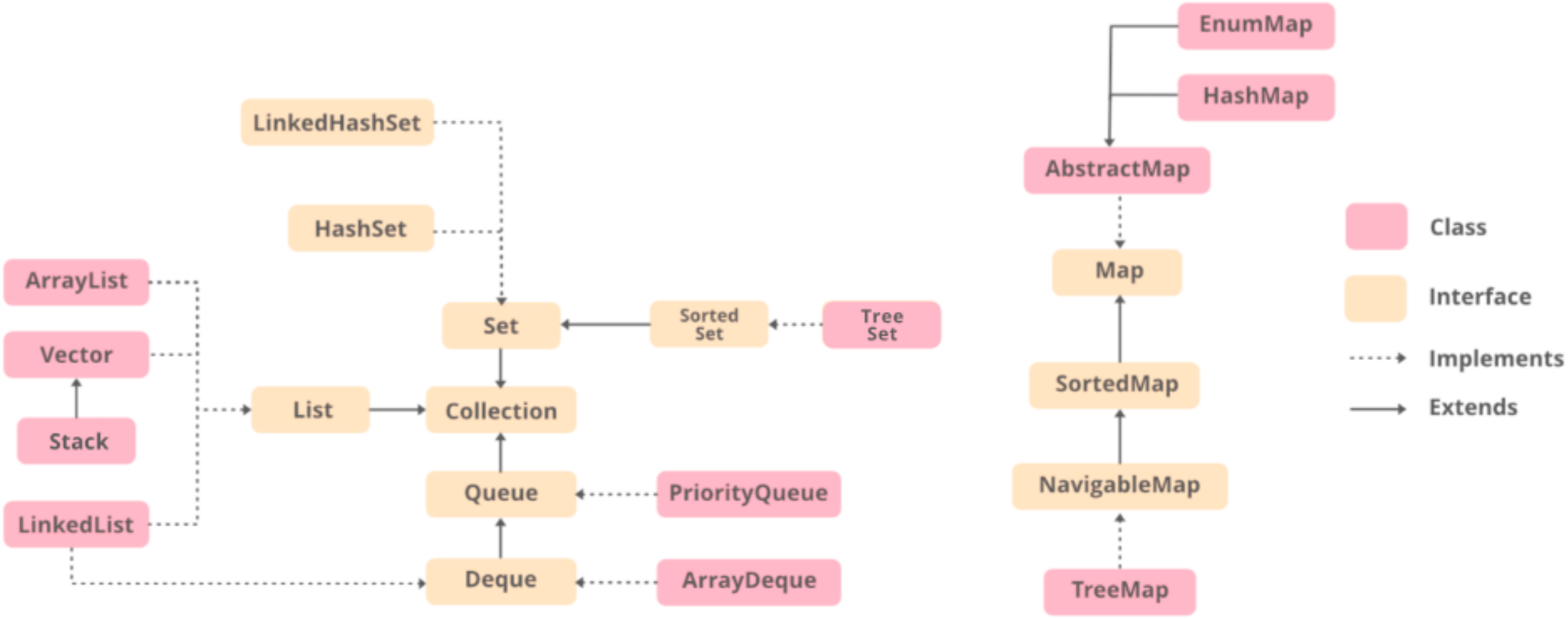
`WeakHashMap` is an implementation of the `Map` interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbage-collected when its key is no longer referenced outside of the `WeakHashMap`. This class provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.

Collections



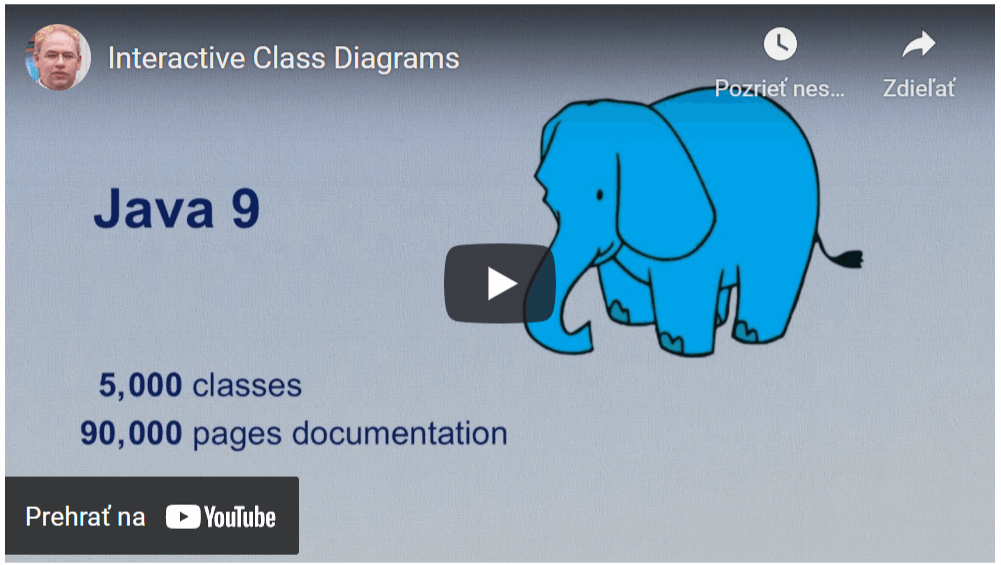
Map


Map
Dictionary
AbstractMap
SortedMap
NavigableMap
Hashtable
HashMap
TreeMap
Properties
LinkedHashMap
Interface
Abstract Class
Class


LinkedHashSet
HashSet
ArrayList
Vector
Stack
LinkedList
Set
Sorted Set
Tree Set
List
Collection
Queue
PriorityQueue
Deque
ArrayDeque
EnumMap
HashMap
AbstractMap
Map
SortedMap
NavigableMap
TreeMap
Class
Interface
Implements
Extends

Content

This website presents enhanced Java class diagrams. The diagrams are interactive and allow access to the complete API documentation. Please see the video for more information.



- About Diagrams — Diagram motivation, explanations and FAQ
- Java 7 — Java 7 SE class diagrams
- Java 8 — Java 8 SE class diagrams
- Java 9 — Java 10 SE class diagrams
- JavaFX 8 — JavaFX 8 class diagrams
- JavaFX 10 — JavaFX 10 class diagrams

- Patreon — Become a Patreon
- PayPal — Donations with PayPal
- Twitter — For feedback and information about updates
- Contact — Personal information
- Legal — Legal notes and Copyright

News

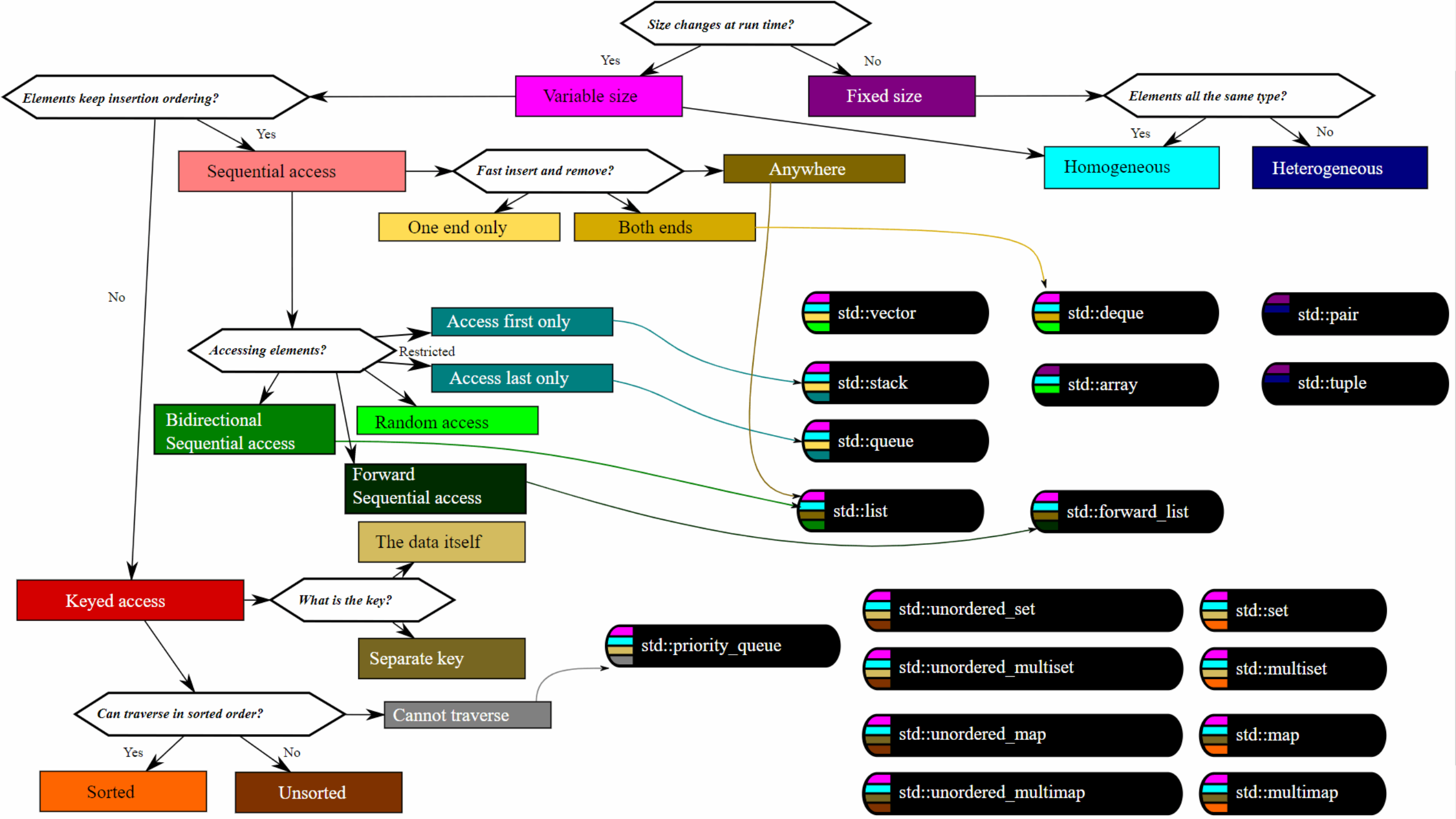
* 22. Apr 2018

Java FX 9
javafx.beans.property ObjectProperty

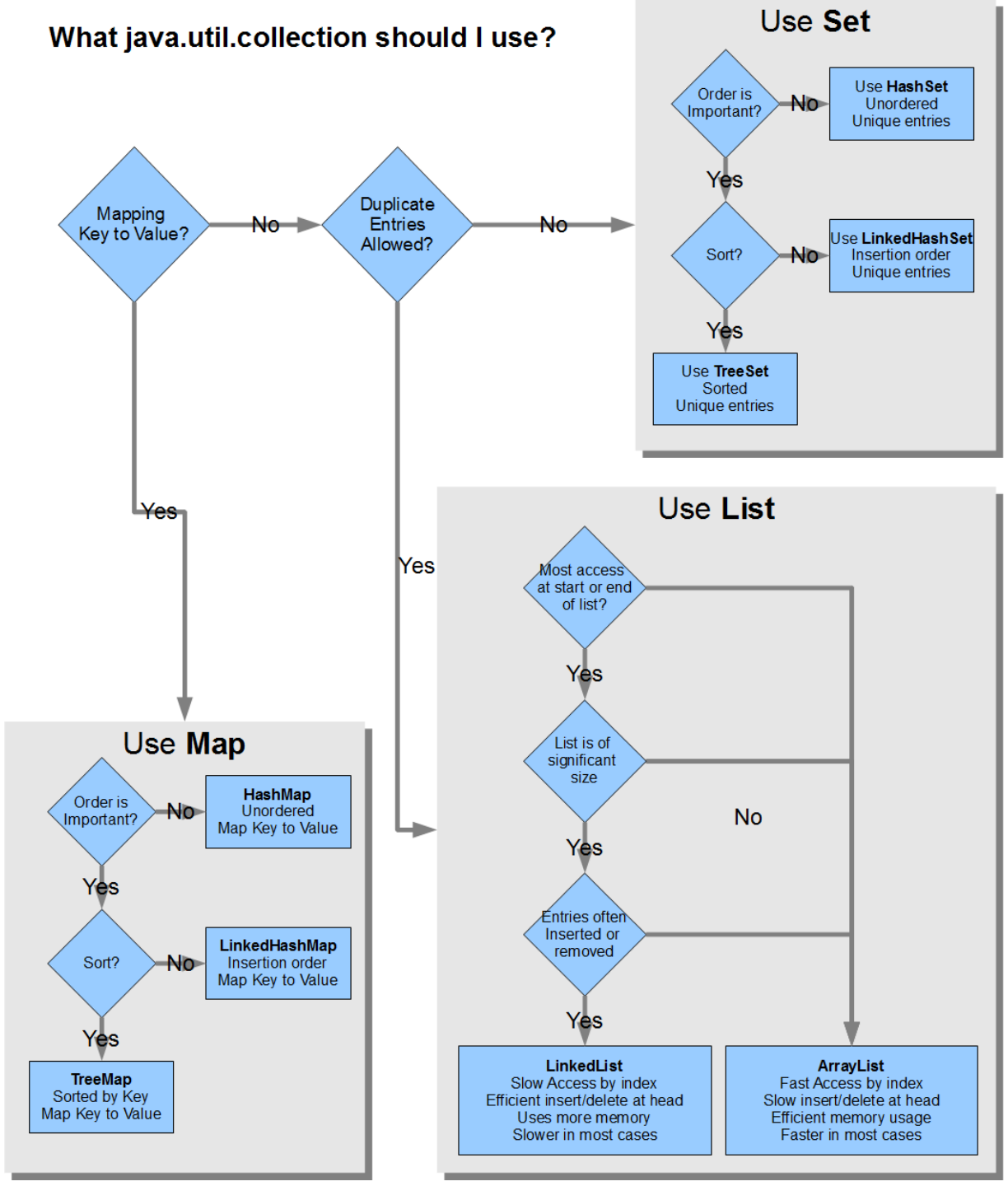
Java 9
javax.crypto.Key

Java FX 9
Control Hierarchy

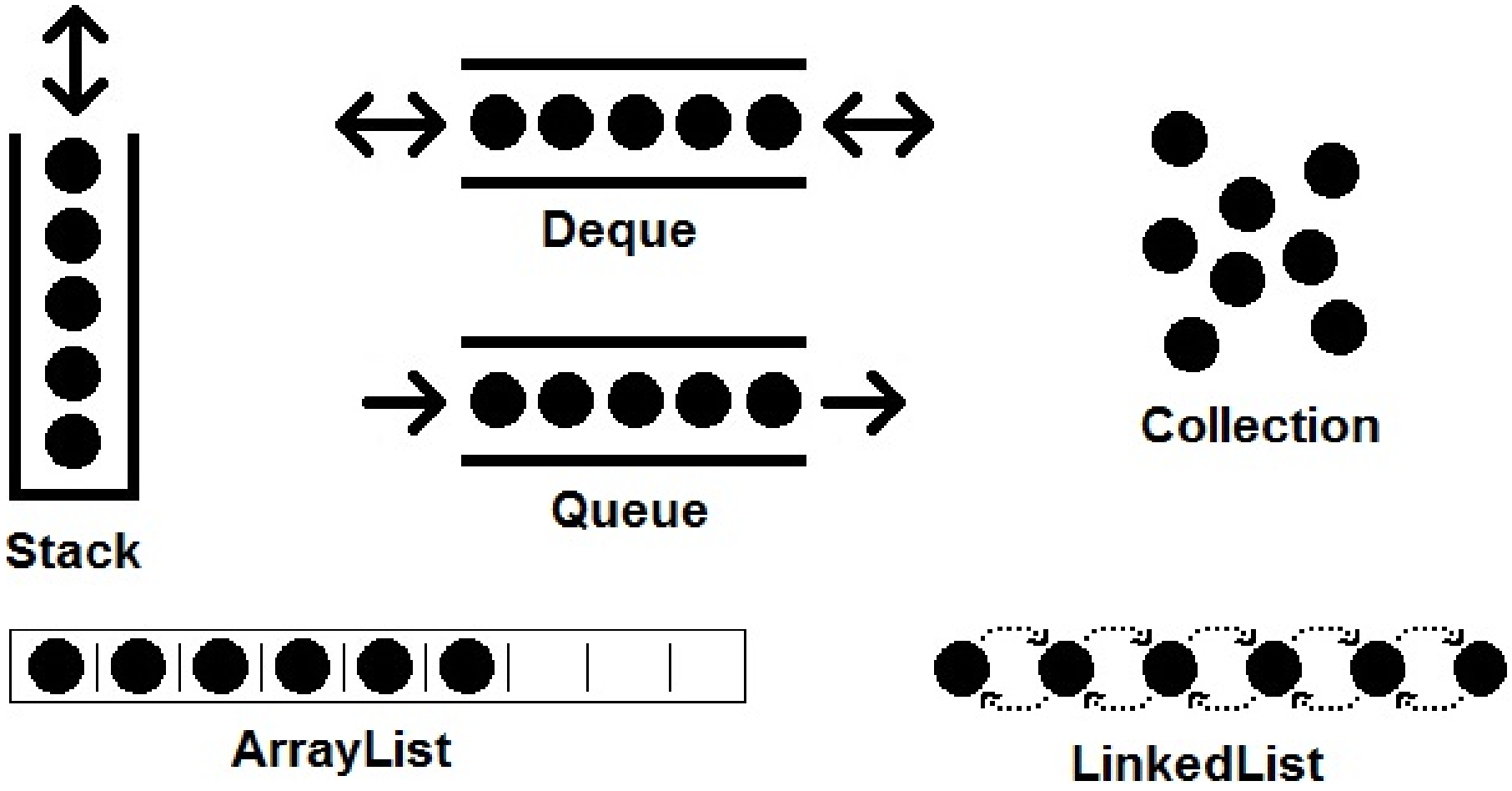
Java 9
javax.swing.JRootPane


Size changes at run time?
Yes
No
Variable size
Fixed size
Elements keep insertion ordering?
Elements all the same type?
Yes
No
Yes
No
Sequential access
Fast insert and remove?
Anywhere
Homogeneous
Heterogeneous
One end only
Both ends
No
Access first only
Accessing elements?
Restricted
Access last only
Bidirectional Sequential access
Random access
Forward Sequential access
The data itself
Keyed access
What is the key?
Separate key
Can traverse in sorted order?
Cannot traverse
Yes
No
Sorted
Unsorted
std::vector
std::deque
std::pair
std::stack
std::array
std::tuple
std::queue
std::list
std::forward_list
std::priority_queue
std::unordered_set
std::set
std::unordered_multiset
std::multiset
std::unordered_map
std::map
std::unordered_multimap
std::multimap

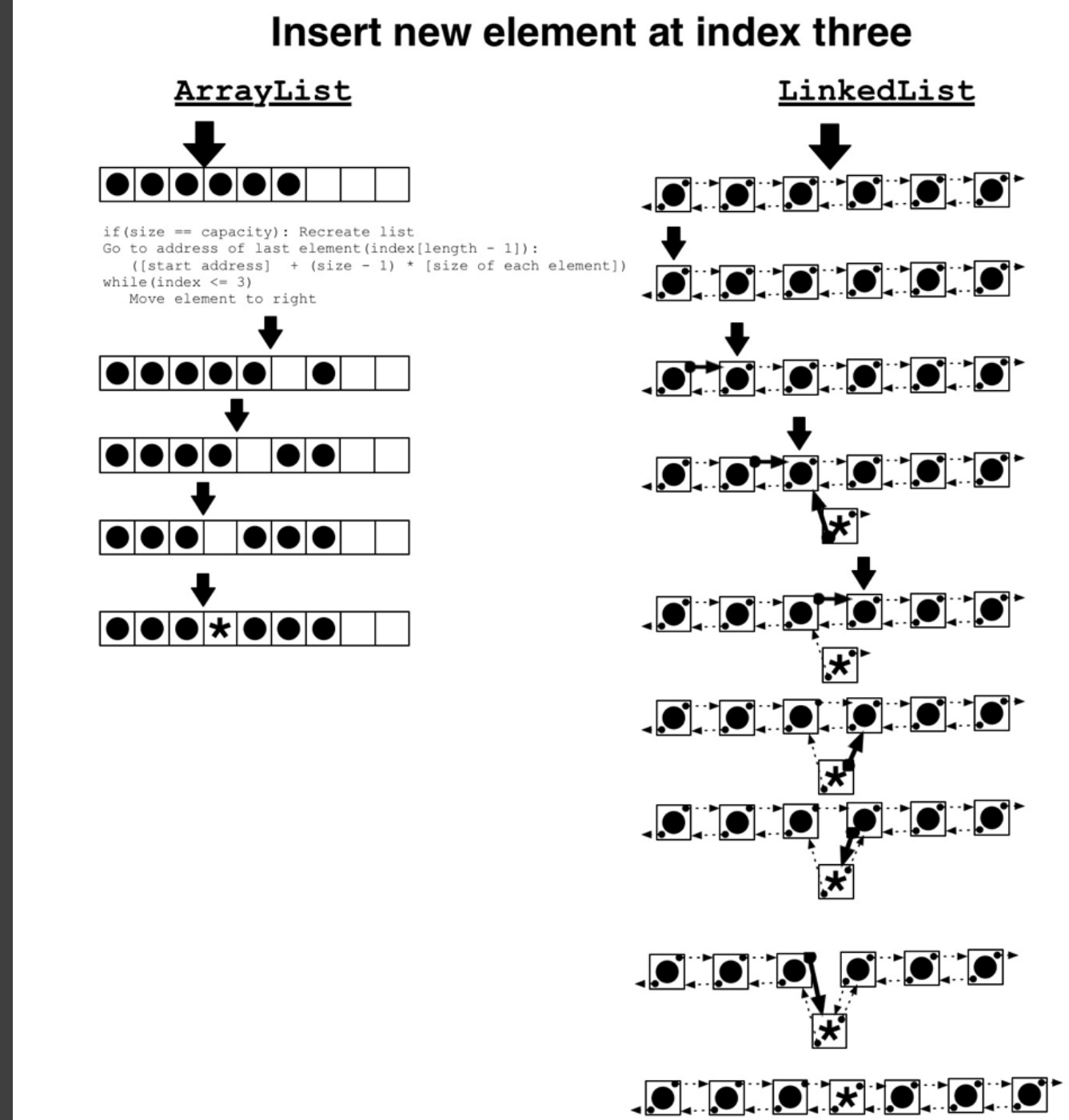
Ktorú kolekciu mám použiť?



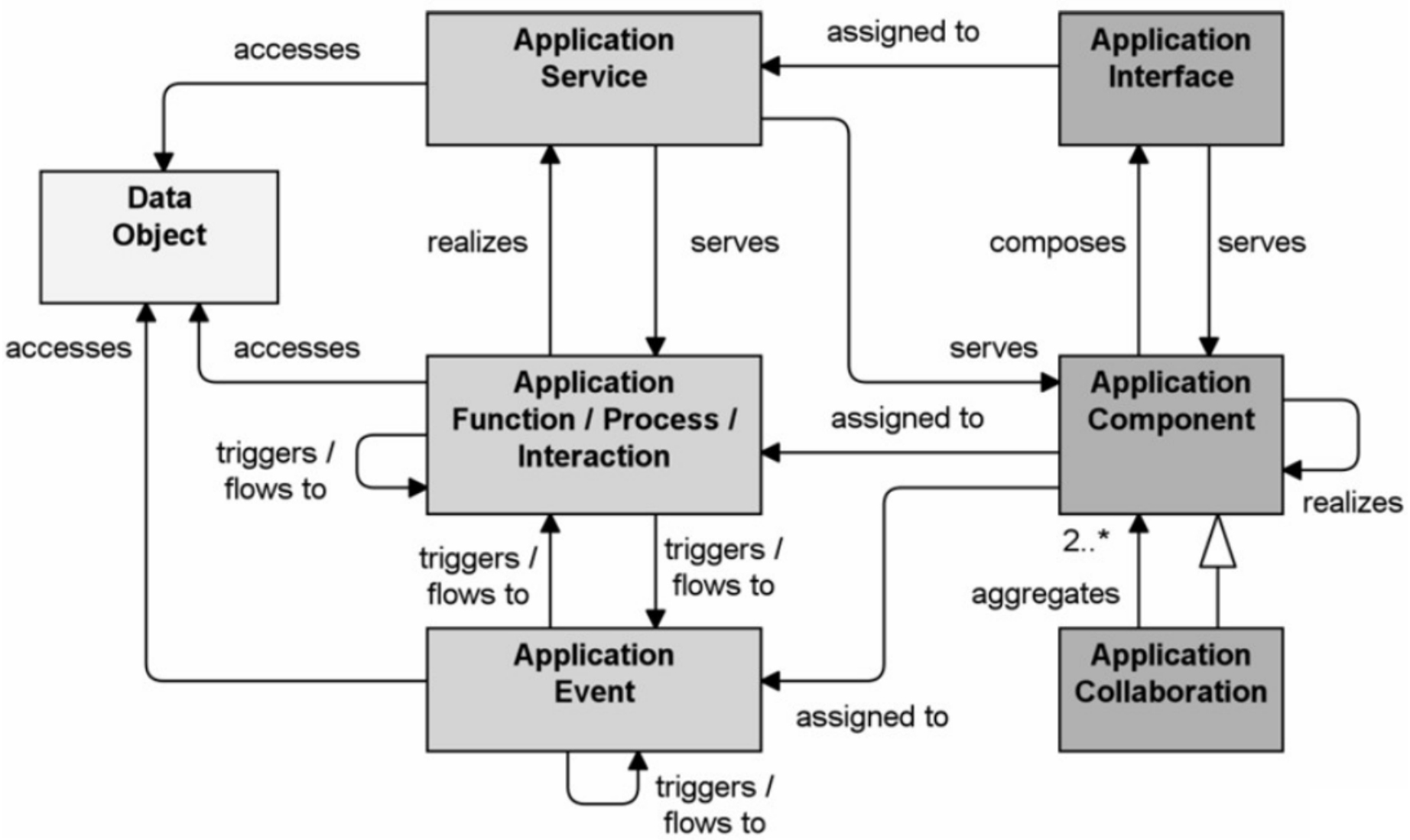
Základné algoritmy za kolekciami



Vkladanie nového elementu

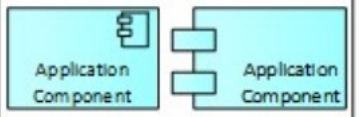
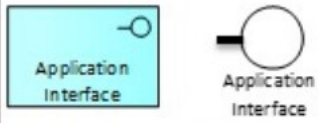
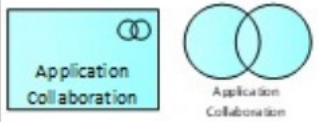
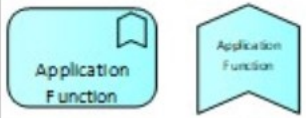
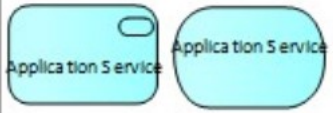
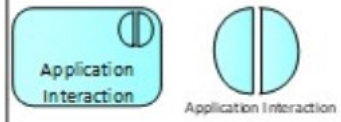
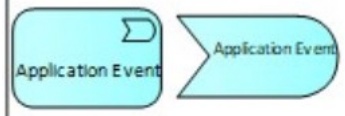


Aplikačná vrstva - Metamodel



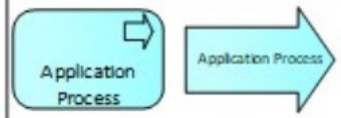
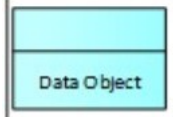
IT ACADEMY

Aplikačná vrstva - Elementy 1

Element	Definice	Aspekt
	An application component represents an encapsulation of application functionality aligned to implementation structure, which is modular and replaceable. It encapsulates its behavior and data, exposes services, and makes them available through interfaces.	Active structure
	An application interface represents a point of access where application services are made available to a user, another application component, or a node.	Active structure
	An application collaboration represents an aggregate of two or more application components that work together to perform collective application behavior.	Active structure
	An application function represents automated behavior that can be performed by an application component.	Behavior
	An application service represents an explicitly defined exposed application behavior.	Behavior
	An application interaction represents a unit of collective application behavior performed by (a collaboration of) two or more application components.	Behavior
	An application event is an application behavior element that denotes a state change.	Behavior

IT ACADEMY

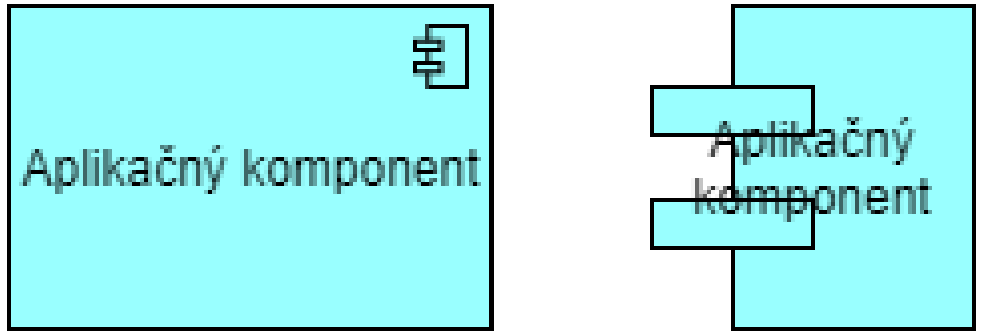
Aplikačná vrstva - Elementy 2

Element	Definice	Aspekt
	An application process represents a sequence of application behaviors that achieves a specific outcome.	Behavior
	A data object represents data structured for automated processing.	Passive Structure

IT ACADEMY

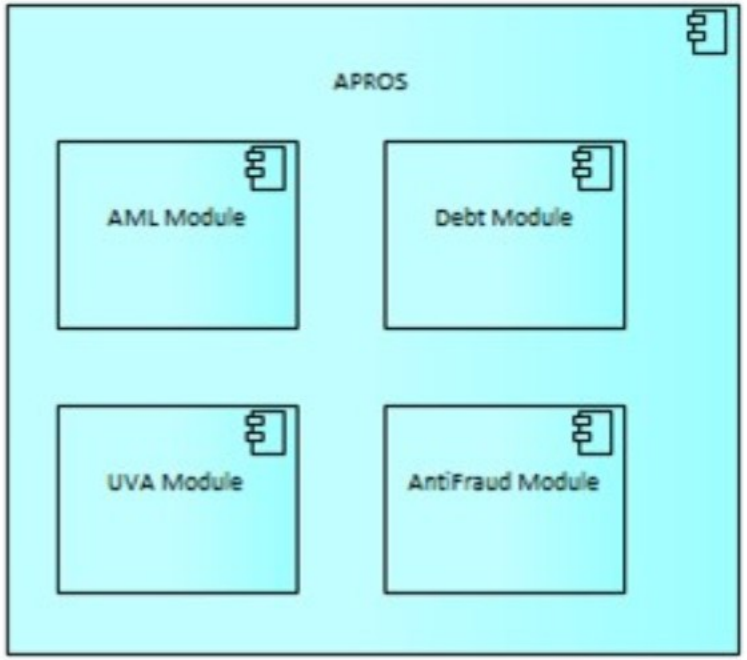
Element Komponent v aplikačnej vrstve (Component)

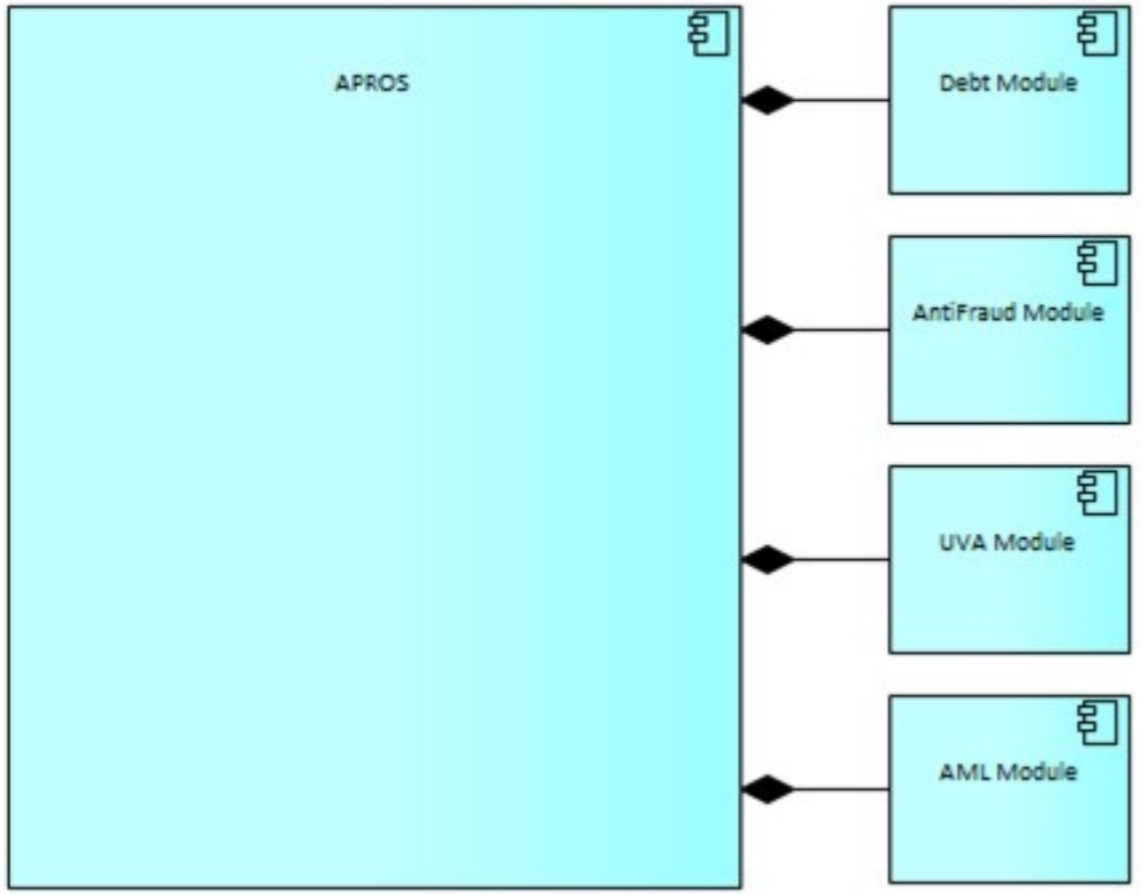
- **Typ**: **aktívny** (Structural Active)
- **Definícia**:
  - Reprezentuje aplikačnú funkcionalitu (z pohľadu implementácie). Komponent je modulárny a nahraditeľný. Zapúzdruje svoje chovanie a dáta, vystavuje do okolného prostredia svoje služby prostredníctvom rozhrania



IT ACADEMY

Element Komponent v aplikačnej vrstve (Component)

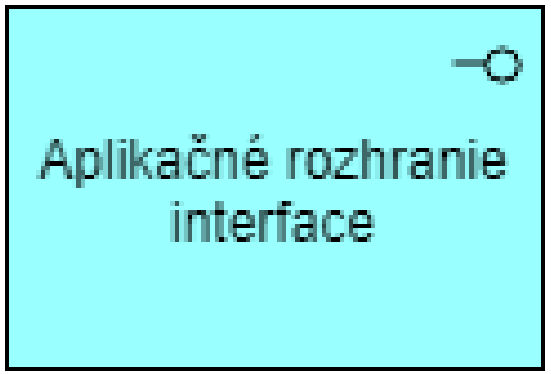




IT ACADEMY

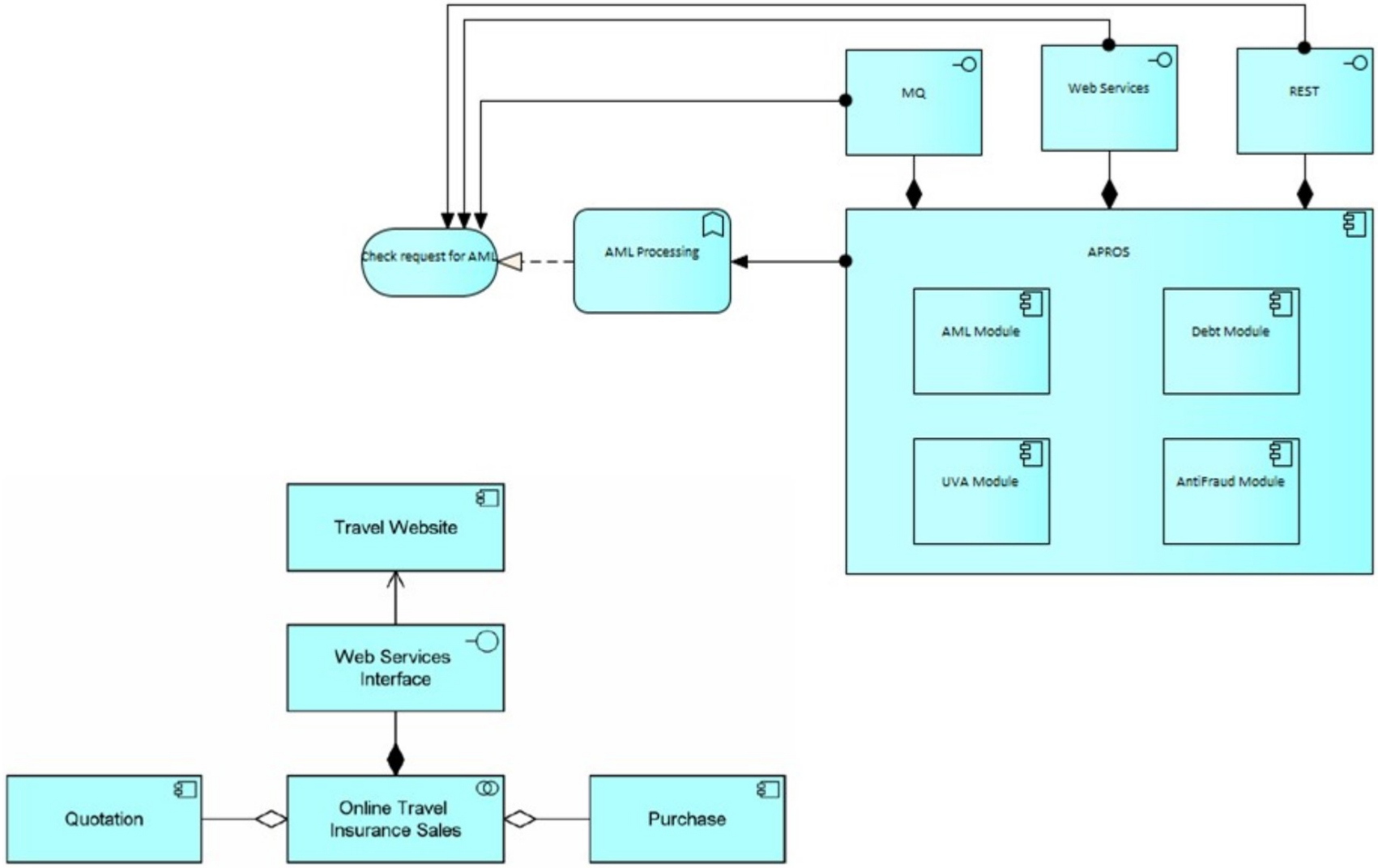
Element Rozhranie v aplikačnej vrstve (Interface)

- **Typ**: **aktívny** (Structural Active)
- **Definícia**:
  - Reprezentuje prístupový bod, kde je aplikačná služba (Application Service) dostupná používateľovi, inej aplikačnej komponente alebo technické uzlu (Technology Node).



IT ACADEMY

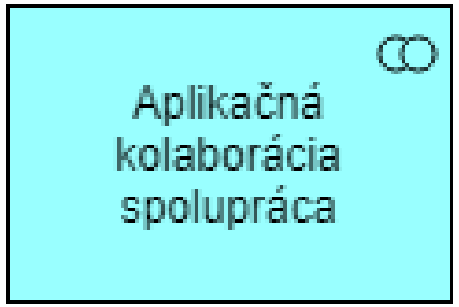
Element Rozhranie v aplikačnej vrstve (Interface)



IT ACADEMY

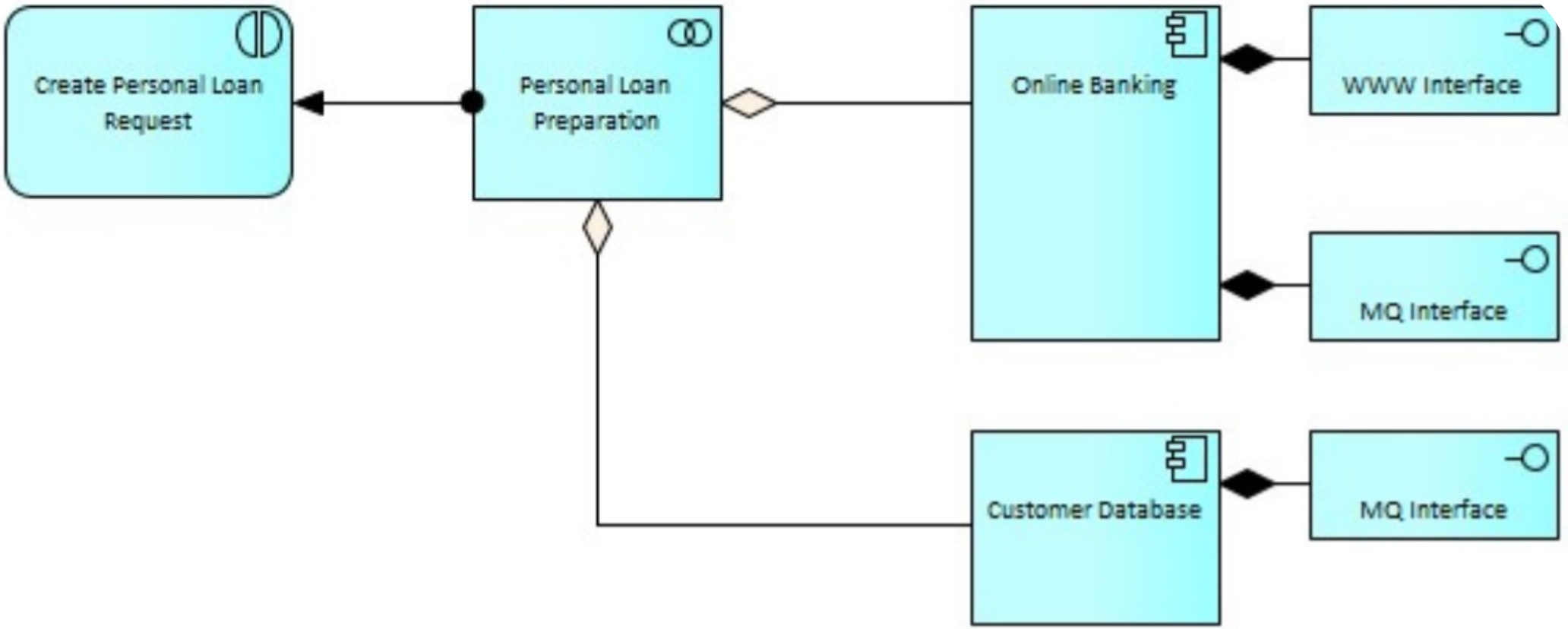
Element Kolaborácia v aplikačnej vrstve (Collaboration)

- **Typ**: **aktívny** (Structural Active)
- **Definícia**:
  - Agreguje 2 alebo viac aplikačných komponent (Application Component), ktoré musia spolupracovať, aby dosiahli spoločného účelu (spoločného chovania - behavior).



IT ACADEMY

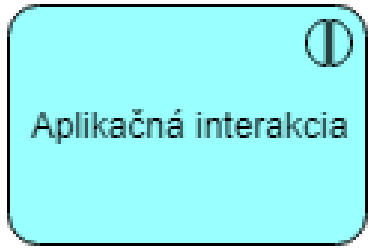
Element Kolaborácia v aplikačnej vrstve (Collaboration)

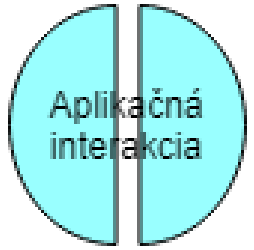


IT ACADEMY

Element Interakcia v aplikačnej vrstve (Interaction)

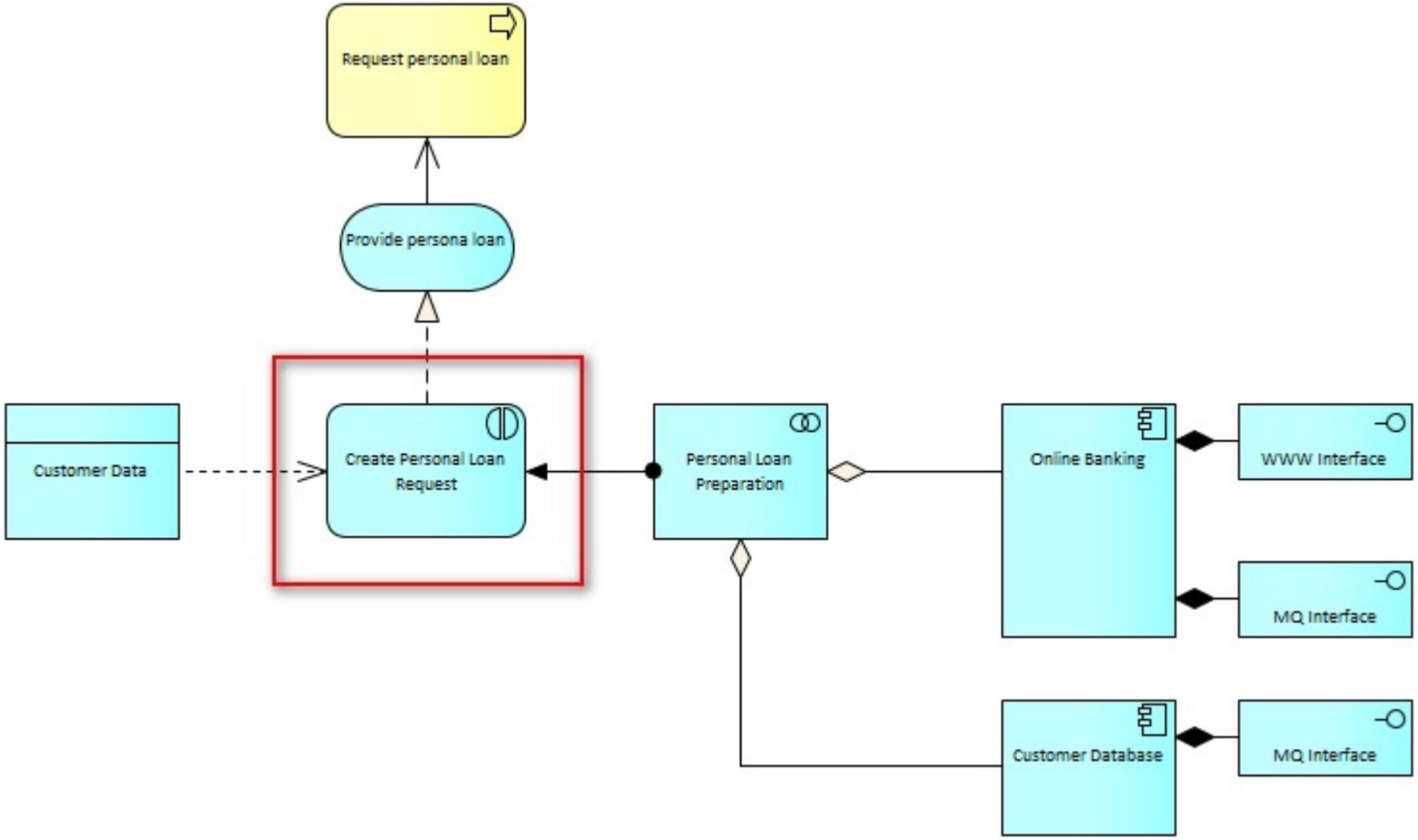
- **Typ**: **správanie** (behavior)
- **Definícia**:
  - Reprezentuje konkrétne spoločné chovanie vykonávané kolaboráciou (Application Collaboration) jednej či viac aplikačných komponent.





IT ACADEMY

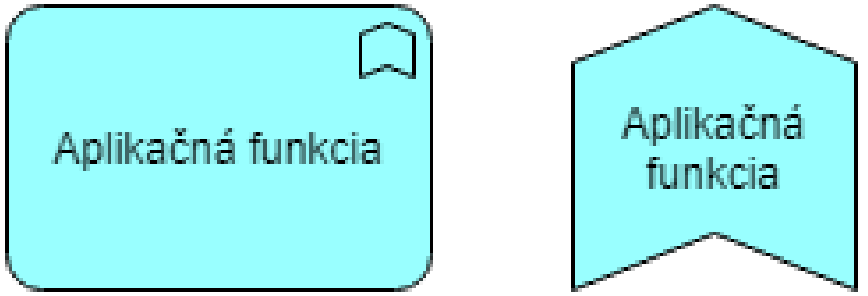
Element Interakcia v aplikačnej vrstve (Interaction)



IT ACADEMY

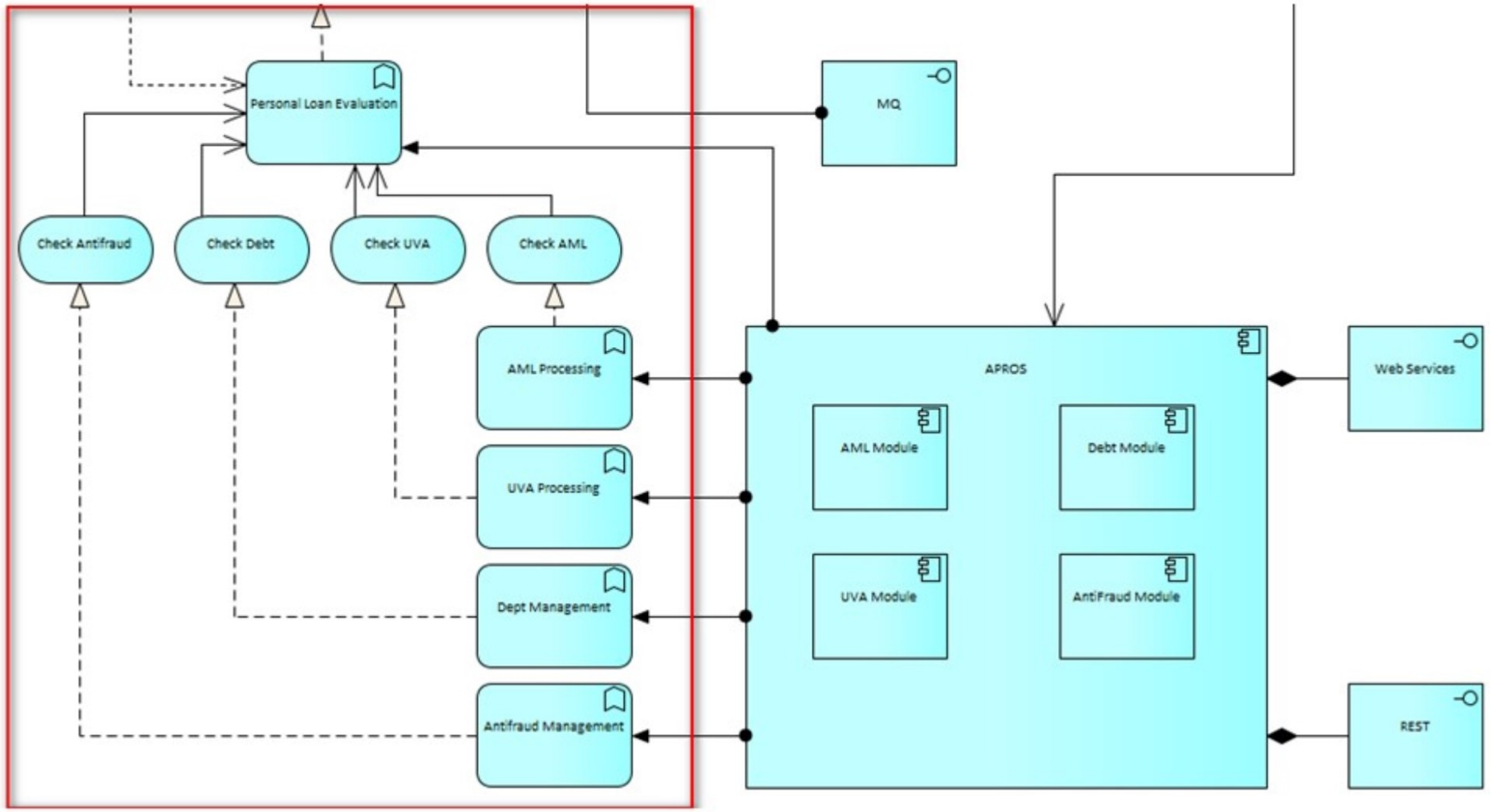
Element Funkcia v aplikačnej vrstve (Function)

- **Typ**: **správanie** (behavior)
- **Definícia**:
  - Reprezentuje automatické chovanie (činnosť), ktorá môže byť vykonávaná niektorou aplikačnou komponentov.



IT ACADEMY

Element Funkcia v aplikačnej vrstve (Function)



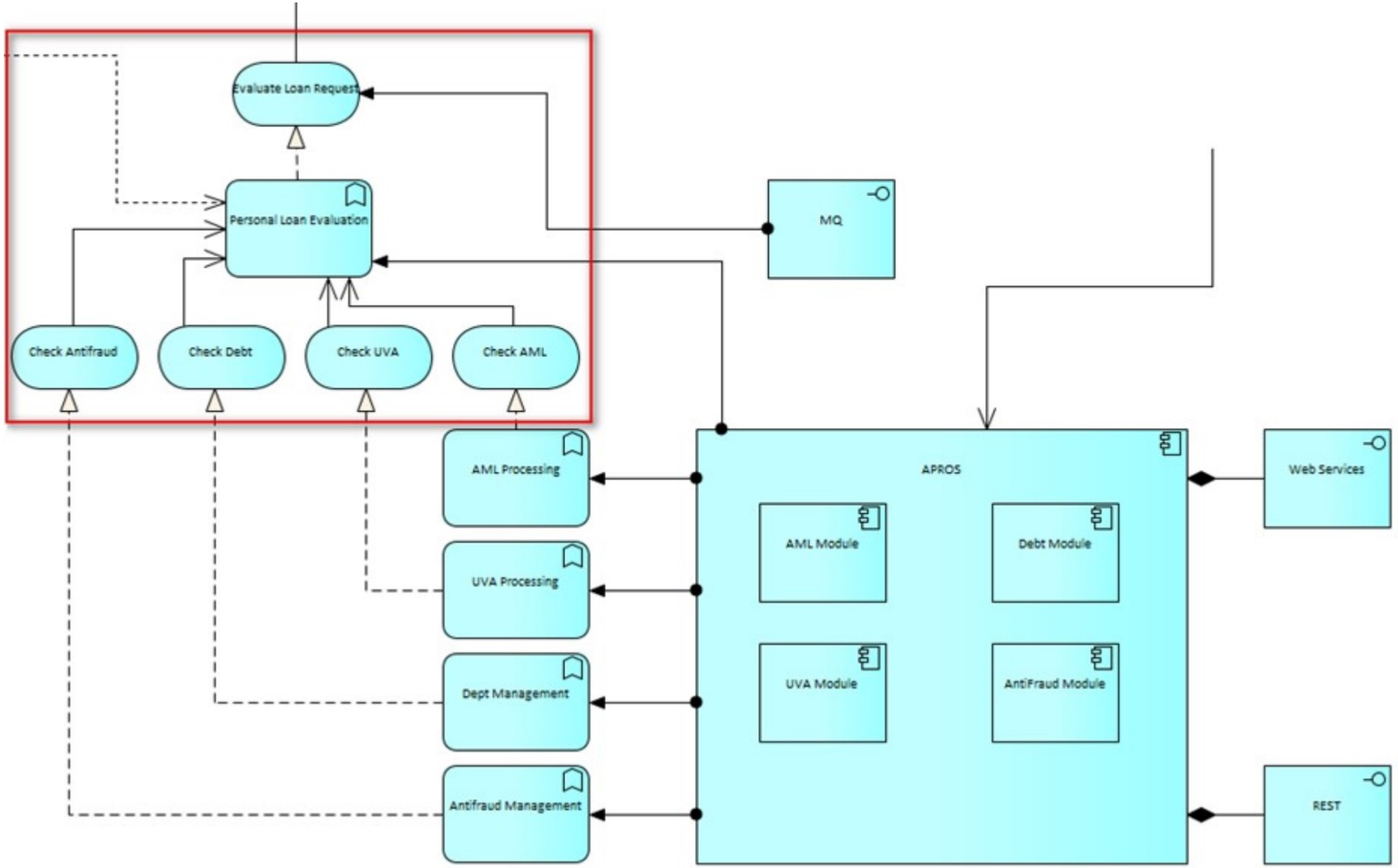
IT ACADEMY

Element Služba v aplikačnej vrstve (Service)

- **Typ**: **správanie** (behavior)
- **Definícia**:
  - Predstavuje explicitne definovaní chovanie (na aplikačnej úrovni), ktoré je vystavené do infraštruktúry.

IT ACADEMY

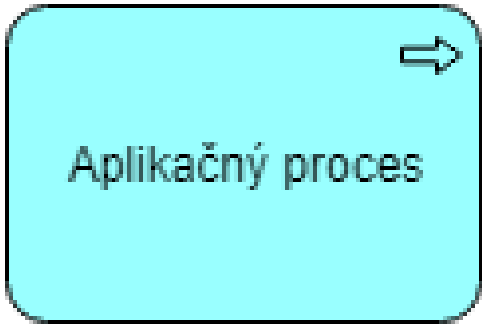
Element Služba v aplikačnej vrstve (Service)

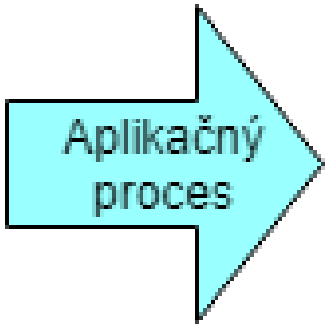


IT ACADEMY

Element Proces v aplikačnej vrstve (Process)

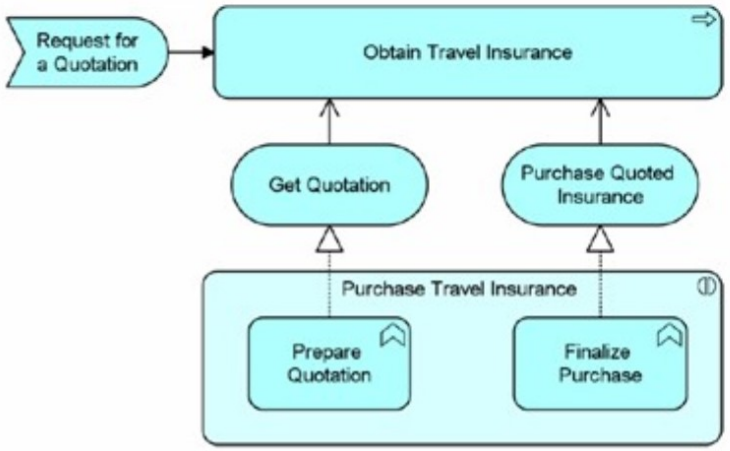
- **Typ**: **správanie** (behavior internal)
- **Definícia**:
  - Reprezentuje sekvenciu akcií (či chovania - behavior) na aplikačnej úrovni, ktoré musia byť splnené pre dosiahnutie požadovaného výstupu.

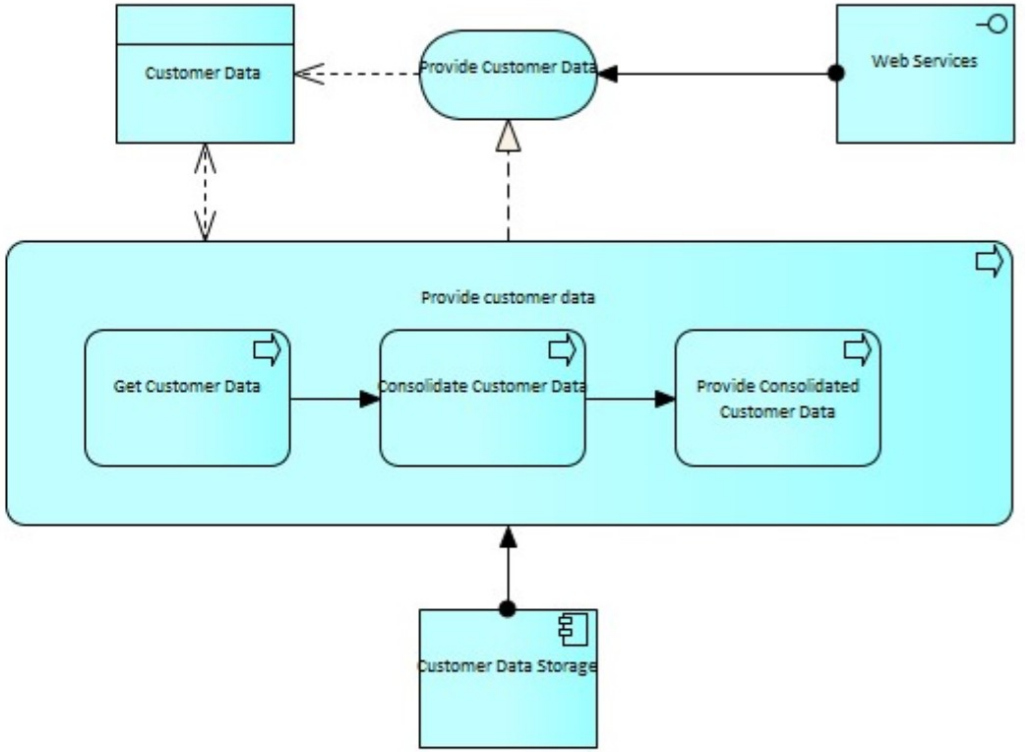




IT ACADEMY

Element Proces v aplikačnej vrstve (Process)

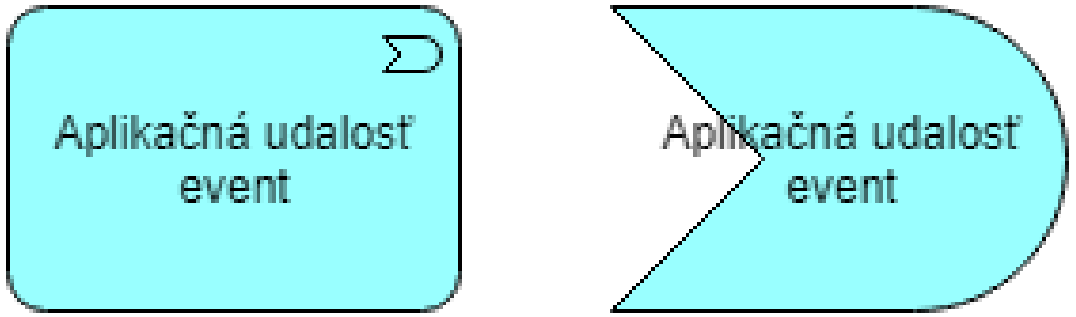




IT ACADEMY

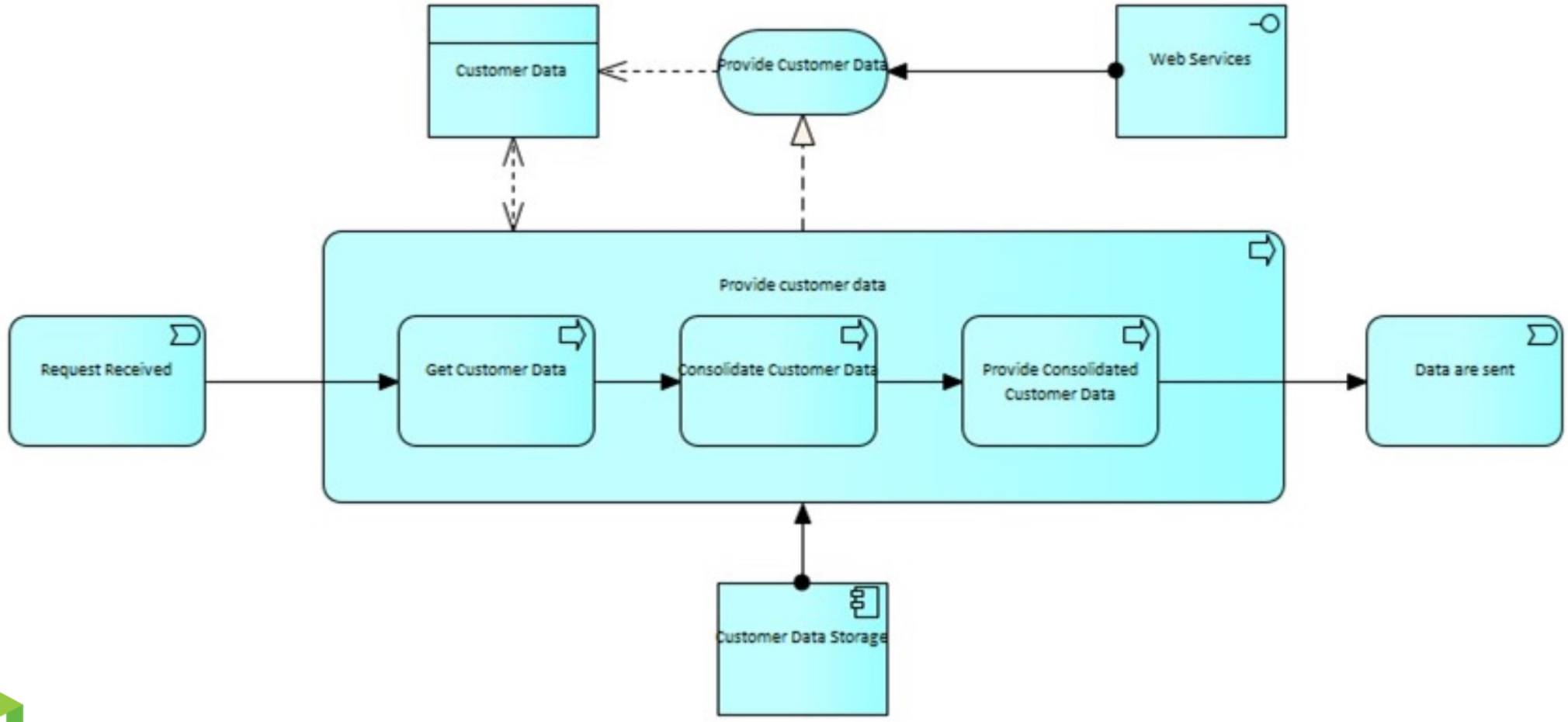
Element Udalosť v aplikačnej vrstve (Event)

- **Typ**: **správanie** (behavior internal)
- **Definícia**:
  - Aplikačné chovanie, ktoré predstavuje zmenu stavu.



IT ACADEMY

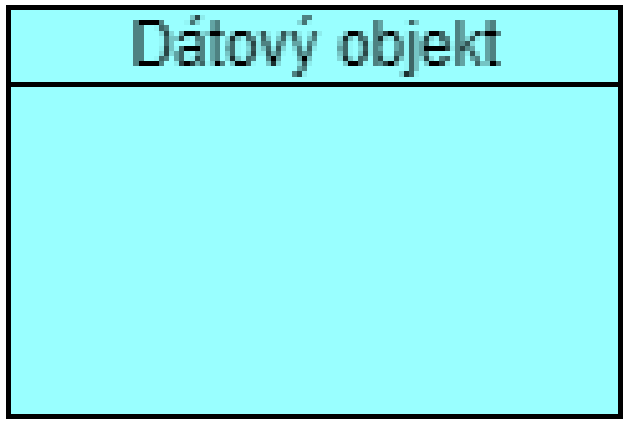
Element Udalosť v aplikačnej vrstve (Event)



IT ACADEMY

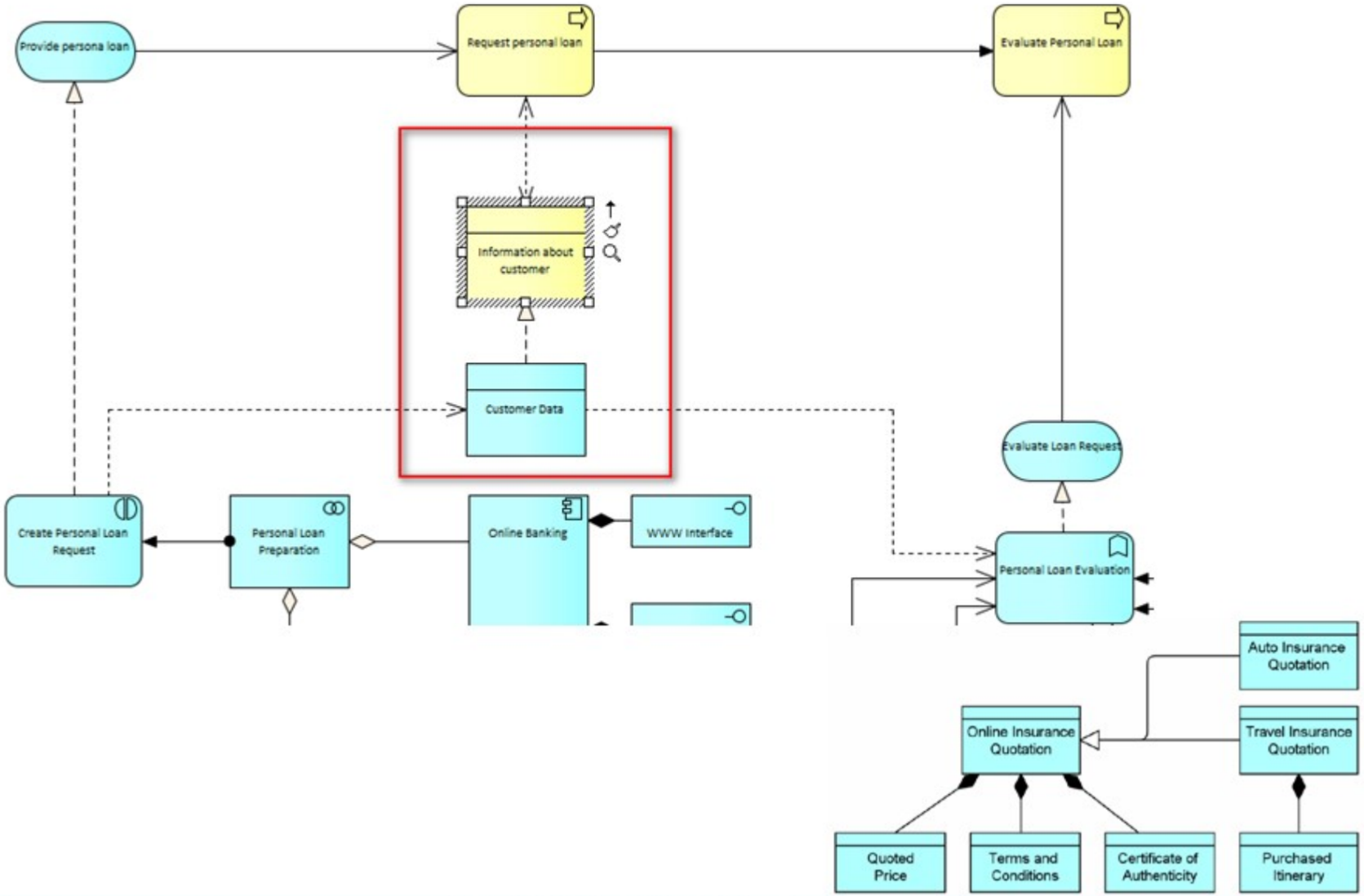
Element Dátový objekt v aplikačnej vrstve (Data Object)

- **Typ**: **pasívna štruktúra** (Passive Structure)
- **Definícia**:
  - Reprezentuje štruktúrované dáta pre automatické spracovanie.



IT ACADEMY

Element Dátový objekt v aplikačnej vrstve (Data Object)



IT ACADEMY