Generics, collections, exception handling

Tamás Ambrus, István Gansperger

Eötvös Loránd University
ambrus.thomas@gmail.com

Generics

Generics in Java

Generics enable types to be parameters when defining classes, interfaces and methods:

- method declarations have formal parameters to re-use the same code with different values,

Generics in Java

Generics enable types to be parameters when defining classes, interfaces and methods:

- method declarations have formal parameters to re-use the same code with different values,
- *generic types/methods* have **type parameters to re-use the same type/method** with different types.

Any number of values **vs** any number of values per types

Example without generics

This is how you should not do:

```java
public class Student {
    private final String name;

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

public class Box {
    private Object item;

    public void setItem(Object item) { this.item = item; }
    public Object getItem() { return item; }
}

// Box b = new Box(); b.setItem(new Student("Tamas"));
// System.out.println(((Student) b.getItem()).getName());
```

Example with generics

This is how you should do:

```java
public class Student {
    private String name;

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

public class Box<T> {
    private T item;

    public void setItem(T item) { this.item = item; }
    public T getItem() { return item; }
}

// Box<Student> b = new Box<>(); b.setItem(new Student("Tamas"));
// System.out.println(b.getItem().getName());
// you can use Box with any reference types, type safe :>
```

Why to use generics?

- Elimination of casts – also gives us more safety from runtime exceptions.
- Stronger type checks at compile time – the compiler can inform us about more errors.
- Enable programmers to implement generic algorithms.
- Not just types, methods can be generic, too.

Type erasure

Generics have no runtime overhead since they are only checked at compile time. After compilation:

- type parameters will be compiled to Object,
- typecasts will be inserted to preserve type safety.

Since there is no runtime information about type parameters we cannot use them like classes. For example, for a type parameter T we cannot write $T\ x = new\ T()$.

```
// backward compatibility :>
```

Type parameters: reference types

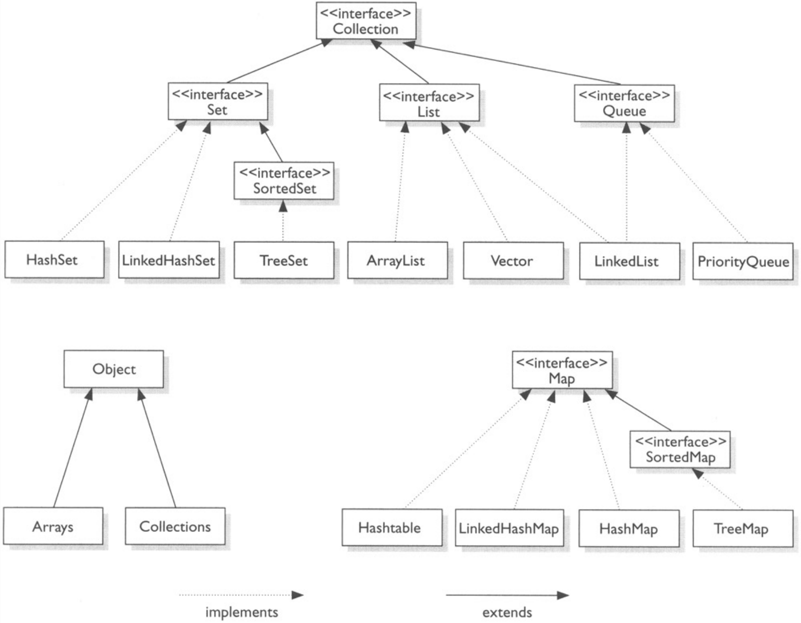
We cannot use primitive types as generic parameters (since they cannot be erased to *Object* at runtime) but they have boxed variants which are reference types:

- Boolean,
- Character,
- Byte, Short, Integer, Long,
- Float, Double.

Converting between these and the original primitive types is called **auto boxing** and **auto unboxing** and is done implicitly by the compiler.

Collections

Collections API



List

The *java.util.List* interface is a subtype of the *java.util.Collection* interface.

It represents an ordered list of objects, meaning you can access the elements in a specific order, and by an index too.
You can also add the same element more than once to a List.

The two prominent implementations are:

- ArrayList: uses an ever-growing array as a backing storage. This is the fastest collection in most cases.
- LinkedList: uses a doubly linked list as backing storage. May be faster if we are doing a lot of insertions.

Set

A collection that contains no duplicate elements. More formally, sets contain no pair of elements *e1* and *e2* such that *e1.equals(e2)*, and at most one null element.

As implied by its name, this interface models the mathematical set abstraction.

The three prominent implementations are:

- HashSet: it is backed by a hash table, does not guarantee that the order will remain constant over time.
- TreeSet: elements are ordered using their natural ordering, or by a *Comparator* provided at set creation time
- LinkedHashSet: it keeps the insertion-order.

Map

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Most used implementations:

- HashMap: fastest but preserves no ordering
- TreeMap: keys are ordered using their natural ordering, or by a *Comparator* provided at map creation time
- LinkedHashMap: it keeps the insertion-order.

The contains method

The *Collection* interface declares a method named *contains* which uses the *equals* method defined on the object.
As the **equals method in Object returns this == obj**, we definitely need to override for types that are stored in collections.

The contains method

The *Collection* interface declares a method named *contains* which uses the *equals* method defined on the object.
As the **equals method in Object returns this == obj**, we definitely need to override for types that are stored in collections.
There are strict rules for the equals method:

- it has to be reflexive, symmetric, transitive and consistent for non-null object references,
- for any non-null reference value x, *x.equals(null)* should return false.

Moreover, as the Javadoc of equals method says, once it is overridden, **hashCode has to be overridden too**.

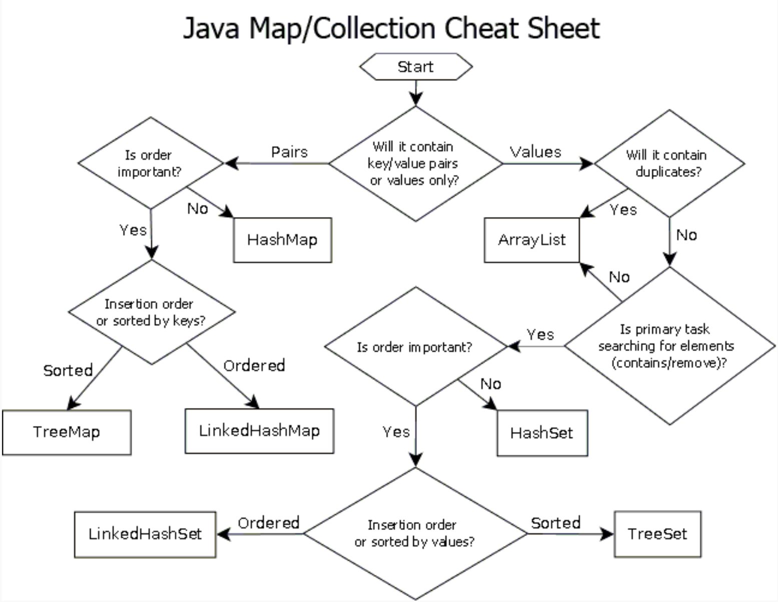
equals and hashCode example

```java
public class Person {
    private final String name; // and set by a constructor

    @Override
    public int hashCode() {
        final int prime = 31;
        return prime + ((name == null) ? 0 : name.hashCode());
    }

    @Override
    public boolean equals(Object obj) { // and NOT equals(Person other)!
        if (this == obj) { // Java programmers' experience: often true
            return true;
        } else if (obj instanceof Person) {
            Person other = (Person) obj;
            return name != null && name.equals(other.name);
        }
        return false; // in most cases a type is not equal to another
    }
}
```

Which collection to use?


Java Map/Collection Cheat Sheet
Start
Will it contain key/value pairs or values only?
Pairs
Values
Is order important?
No
HashMap
Yes
Insertion order or sorted by keys?
Sorted
TreeMap
Ordered
LinkedHashMap
Will it contain duplicates?
Yes
ArrayList
No
Is primary task searching for elements (contains/remove)?
No
Yes
Is order important?
No
HashSet
Yes
Insertion order or sorted by values?
Ordered
LinkedHashSet
Sorted
TreeSet

Exceptions

Exception handling

An exception is an event that occurs during the execution of a program that **disrupts the normal flow** of instructions.

When an error occurs within a method, the method:

- creates an object (which is called *exception object*) that will contain information about the error including
    - its type,
    - the state of the program when the error occurred,
- hands it off to the runtime system.

This mechanism is called: **throwing an exception**.

Exception handling

There are two types of exceptions.

Checked exceptions:

- are subclasses of *Exception*
- represent invalid conditions that have to be known and handled by the programmer
  - invalid user input
  - database problems
  - network outages
  - absent files
- methods have to handle or propagate all checked exceptions

Exception handling

Unchecked exceptions:

- are subclasses of *RuntimeException*
- represent defects in the program (bugs)
  - "Unchecked runtime exceptions represent conditions that, generally speaking, reflect errors in your program's logic and cannot be reasonably recovered from at run time."
- methods don't have to handle or propagate unchecked exceptions (and they don't do)

Exception propagation

When using checked exceptions one must specify in the method signature if the method doesn't handle the exception but rather propagates it.
This can be done with the *throws* keyword.

```
public void throwException() throws Exception {
    throw new Exception();
}
```

Example

```java
public class Stack<T> {
    private T[] elements;
    private int lastIndex;
    // ...

    public T pop() throws EmptyStackException {
        if (elements.length == 0) {
            throw new EmptyStackException(
                "Cannot invoke pop on an empty stack");
        } else {
            lastIndex--;
        }
        return elements[lastIndex];
    }
}

public class EmptyStackException extends Exception {
    public EmptyStackException(String message) {
        super(message);
    }
}
```

Example

```java
public static void main(String[] args) {
    Stack<Integer> intStack = new Stack<>();
    try {
        intStack.pop();
        intStack.push(2);
    } catch (EmptyStackException ex) {
        ex.printStackTrace();
    } finally {
        System.out.println("ALWAYS get printed");
    }
    // ...
}
```

References

- https://docs.oracle.com/javase/tutorial/java/generics/types.html
- https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html
- https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html#equals-java.lang.Object-
- https://docs.oracle.com/javase/9/docs/api/java/lang/Exception.html