Java
Lecture 7 - Arrays and Collections

IT Learning &
Outsourcing Center

www.pragmatic.bg

Lector: Milen Penchev
Skype: donald8605
E-mail: milen.penchev@gmail.com

Agenda

- Arrays
- Collections
- Set
- List
- Map

Problem

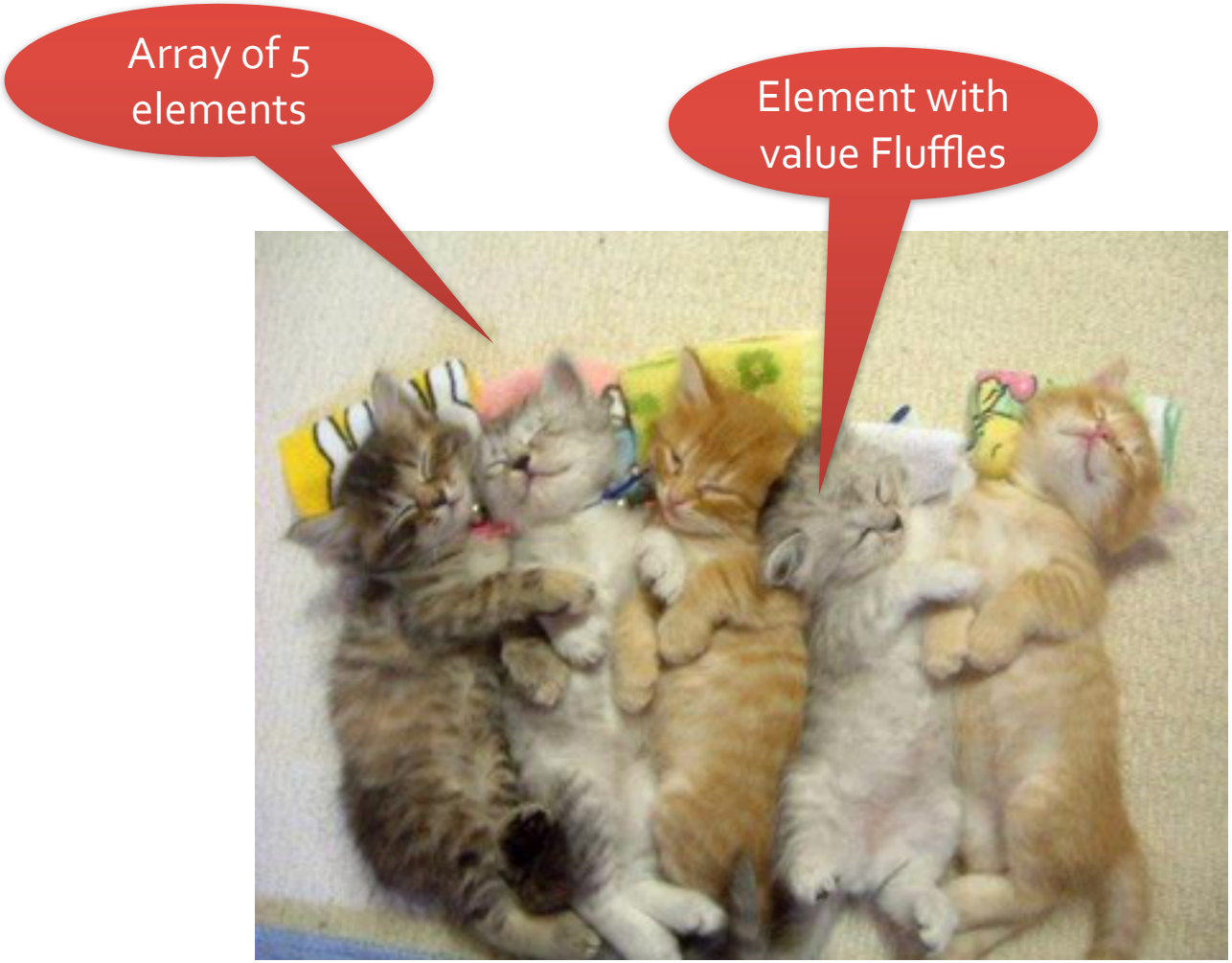
- Define more than one variable for similar purpose
- ***Example:***
  Grades of a student group – define 30 variables for them
- ***Solution:***
  Define 30 variables of type double to hold the information

Is this so rational?

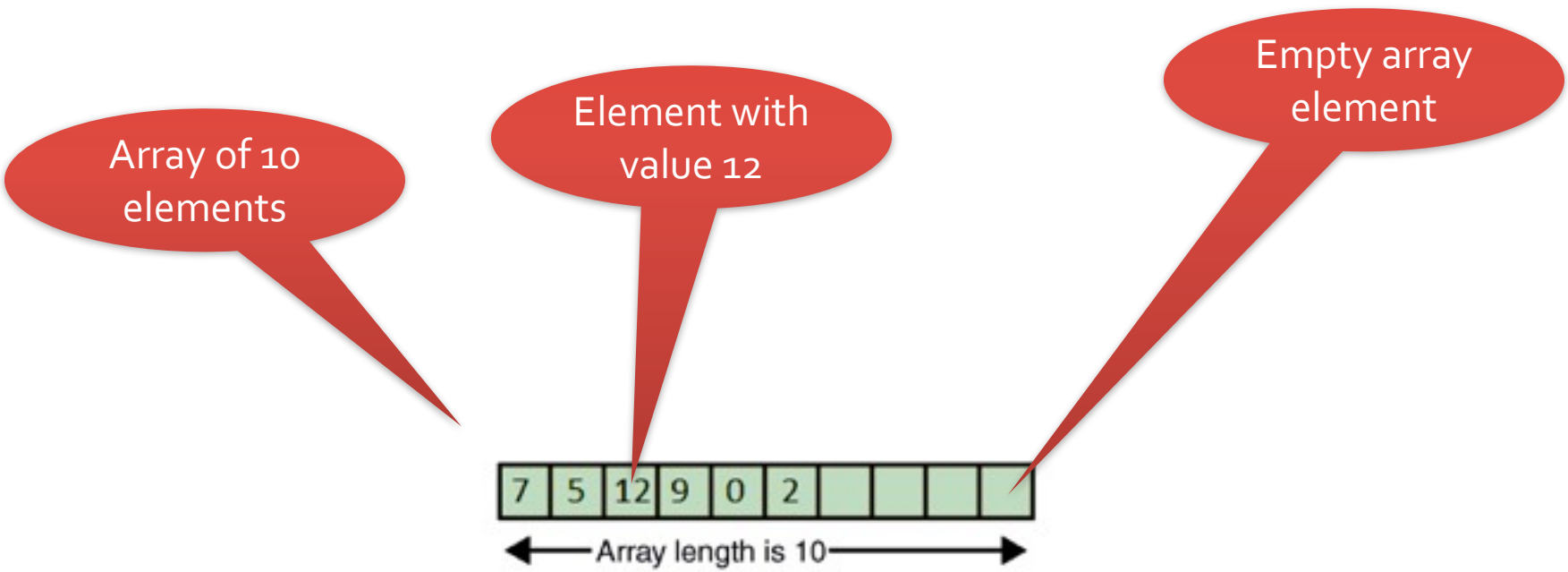
What's an array?

- An array is a sequence of elements
- Arrays keep variables of only one type
- The order of the elements remains the same
- Arrays have a fixed length
- The access to the elements is direct
- The elements are accessed through an index

What's an array?



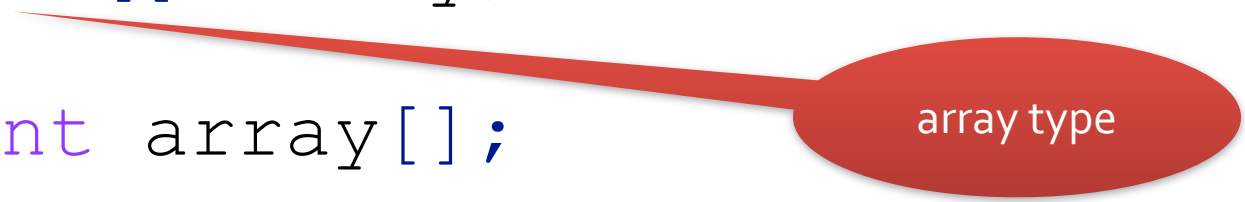
What's an array?



Declaration and initialization

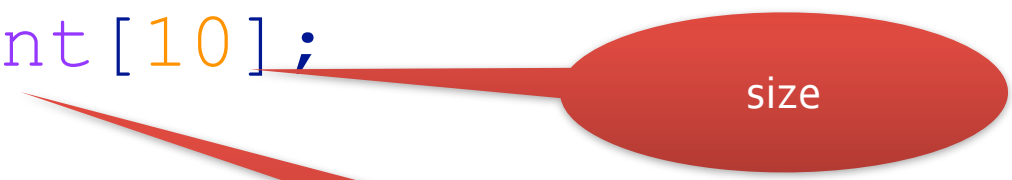
- Declaration

```
int[] array;

int array[];
```



- Initialization

```
array = new int[10];
```

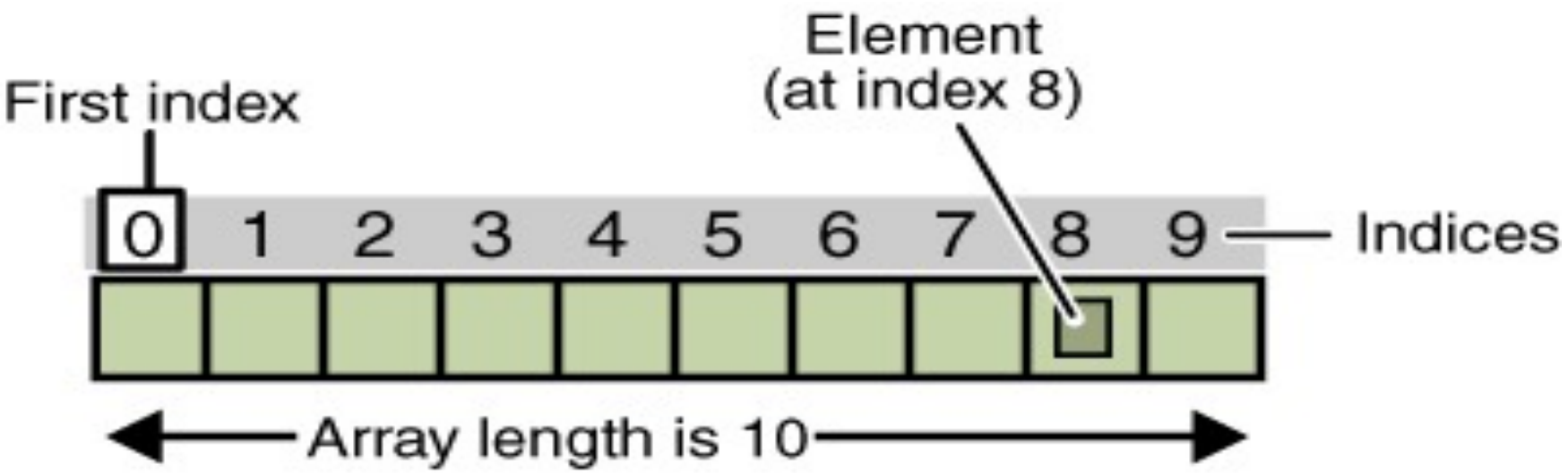


- Declaration and initialization


```
int[] array = new int[10];
int[] array = { 5, 7, -2, 12, 0, 4 };
```

Accessing the elements

- Elements are accessed by index
- The index of the first is 0
- The index of the last is equal the length – 1
- The elements can be read and changed



Accessing the elements

- array[i] returns the value of element with index i

```
System.out.println(array[0]);
//prints the value of the first array element

System.out.println(array[1]);
//prints the value of the second array element


array[2] = 100;
//changes the value of the third element
```

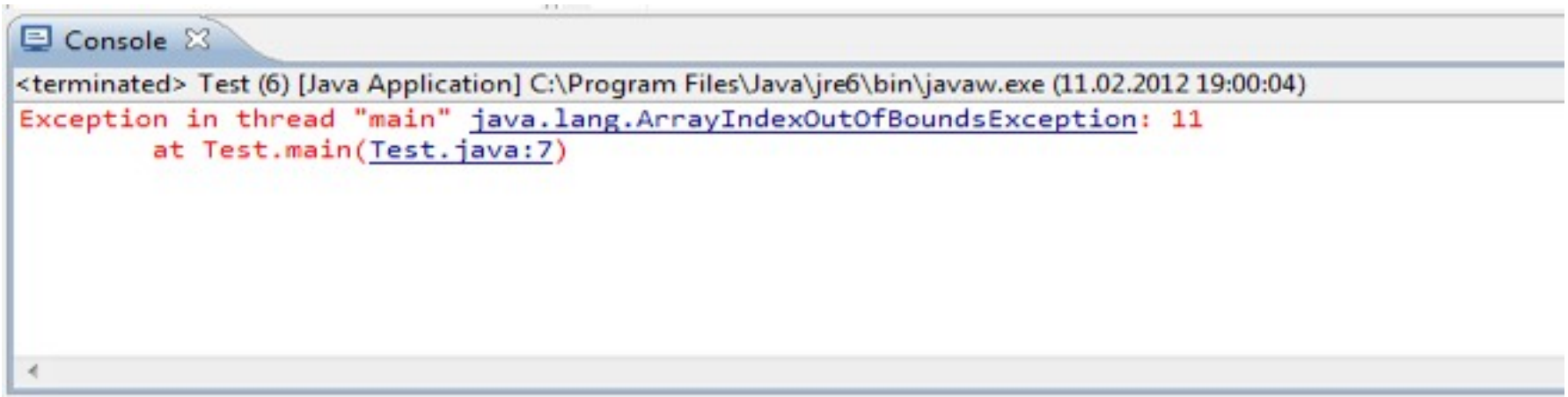
Out of bounds

- array.length returns the length

```
System.out.println(array.length);
```

- Getting an element beyond the size will result in a runtime error

```
array[11] = 20;
```



Iterating the array with for loop

- Normally we're using loops to iterate over an array
- The most common case is using a **for loop**

```
public static void main(String[] args) {
      int[] array = new int[10];
      for (int i = 0; i < array.length; i++) {
            array[i] = 7;
      }
}
```

Iterating the array with while loop

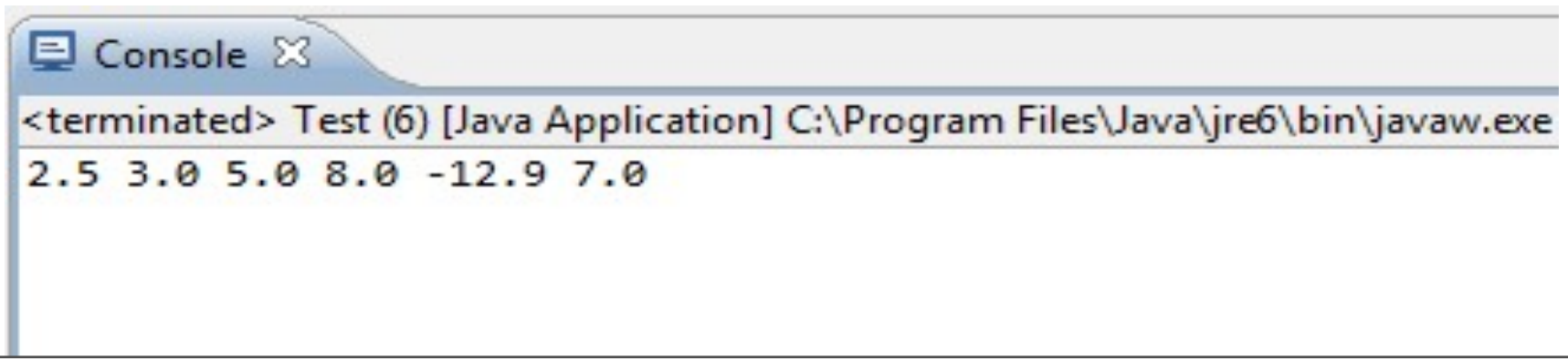
- You can iterate array with **while loop and any other**

```
public static void main(String[] args) {
	int[] array = new int[10];
	int i = 0;
	while (i < array.length) {
		array[i] = 7;
		i++;
	}
}
```

Printing to console

- The array is iterated
- The value of the current element is printed using System.out.print()

```
double[] array = { 2.5 ,3, 5, 8, -12.9, 7.0 };

for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + " ");
}
```



Reading from console

- The array is iterated
- Use scanner to read the value from the console
- Assign the read value to the current element

```
public static void main(String[] args) {
	//declaration and initialization
	int[] array = new int[10];

	//create Scanner
	Scanner sc = new Scanner(System.in);

	//Iterate with for loop and read value for each
	//element from console
	for (int i = 0; i < array.length; i++) {
		System.out.println("Enter value:");
		array[i] = sc.nextInt();
	}
} //ArrayReadingFromConsole.java in code examples
```

Comparing arrays

- Arrays are referred types and can't be compared using **== operator**
- To compare two arrays, you have to iterate them and compare their elements respectively.

- Let's give it a try!

```
double[] array = { 2.5 , 3, 5.8 };
double[] array2 = new double[3];
array2[0] = 2.5;
array2[1] = 3;
array2[2] = 5.8;
...
```

- Lets take a look in ArrayCompare.java in code examples

Copying arrays

- `int[] newArray = oldArray;`
- The line above is not really what you want
- What would be the result of this code?

```
public static void main(String[] args) {
        int[] oldArray = { 1, 2, 3};
        int[] newArray = oldArray;

        oldArray[0] = -10;
        System.out.println(newArray[0]);
}
```

- Lets demonstrate System.arraycopy() method ArrayCopyDemo.java in code examples

Sorting arrays

- Bubble sort
- Selection sort
- Quick sort

Multidimentional Arrays

- Have more than one dimension (2, 3, 4, …)
- The 2-dimensional arrays are called matrices
- A matrix is an array in which each element is an array

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Creating and iterating matrix

- The multidimensional arrays use the same concept as an ordinary arrays

```
public static void main(String[] args) {
        int[][] matrix = new int[3][4];

        for (int i = 0; i < matrix.length; i++) {
                for (int j = 0; j < matrix[0].length; j++) {
                        matrix[i][j] = 10;
                }
        }
        matrix[0][0] = 1;
        matrix[2][3] = 100;
}
```

Creating the array

Setting value for top left element

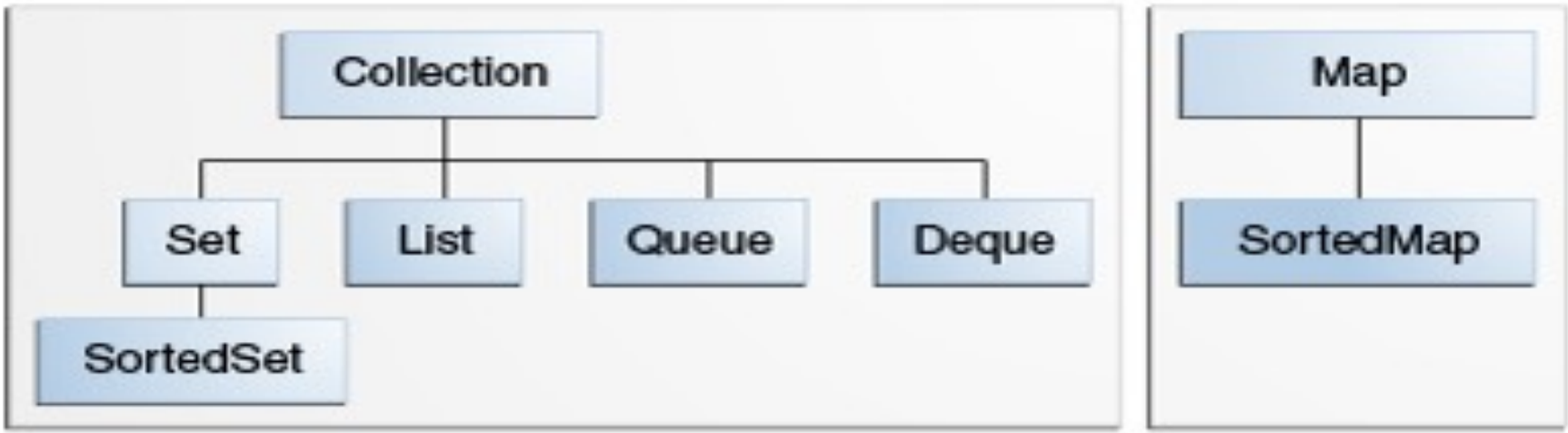
Setting value for bottom right element

Creating and iterating matrix

Collections - Introduction

- 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).

Collection Interfaces



The Collection Interface


```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Collection methods

- how many elements are in the collection (*size, isEmpty*)
- to check whether a given object is in the collection (*contains*)
- to add and remove an element from the collection (*add, remove*)
- to provide an iterator over the collection (*iterator*).

for-each construct

- The *for-each* construct allows you to concisely traverse a collection or array using a for loop. The following code uses the *for-each* construct to print out each element of a collection on a separate line.

```
for (Object o : collection) {
    System.out.println(o);
}
```

- Let's take a look ForEachExample.java in code examples

Iterator interface

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

- The *hasNext* method returns true if the iteration has more elements, and the next method returns the next element in the iteration.
- The remove method removes the last element that was returned by next() from the underlying Collection. The remove method may be called only once per call to next() and throws an exception if this rule is violated.
- Lets take a look on IteratorExample.java file in code examples

for-each vs. Iterator

- Use *Iterator* instead of the *for-each* construct when you need to:
  - Remove the current element. The for-each construct hides the *iterator*, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
  - Iterate over multiple collections in parallel.

Collection interface Bulk Operations

- *containsAll* — returns true if the target Collection contains all of the elements in the specified Collection.
- *addAll* — adds all of the elements in the specified Collection to the target Collection.
- *removeAll* — removes from the target Collection all of its elements that are also contained in the specified Collection.
- *retainAll* — removes from the target Collection all its elements that are *not* also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
- *clear* — removes all elements from the Collection.

The *addAll*, *removeAll*, and *retainAll* methods all return true if the target Collection was modified in the process of executing the operation.

Collection Interface Array Operations

- The *toArray* methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a Collection to be translated into an array. The simple form with no arguments creates a new array of Object. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

```
Object[] a = c.toArray();
```

- Suppose that c is known to contain only strings (perhaps because c is of type Collection<String>). The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

```
String[] a = c.toArray(new String[]);
```

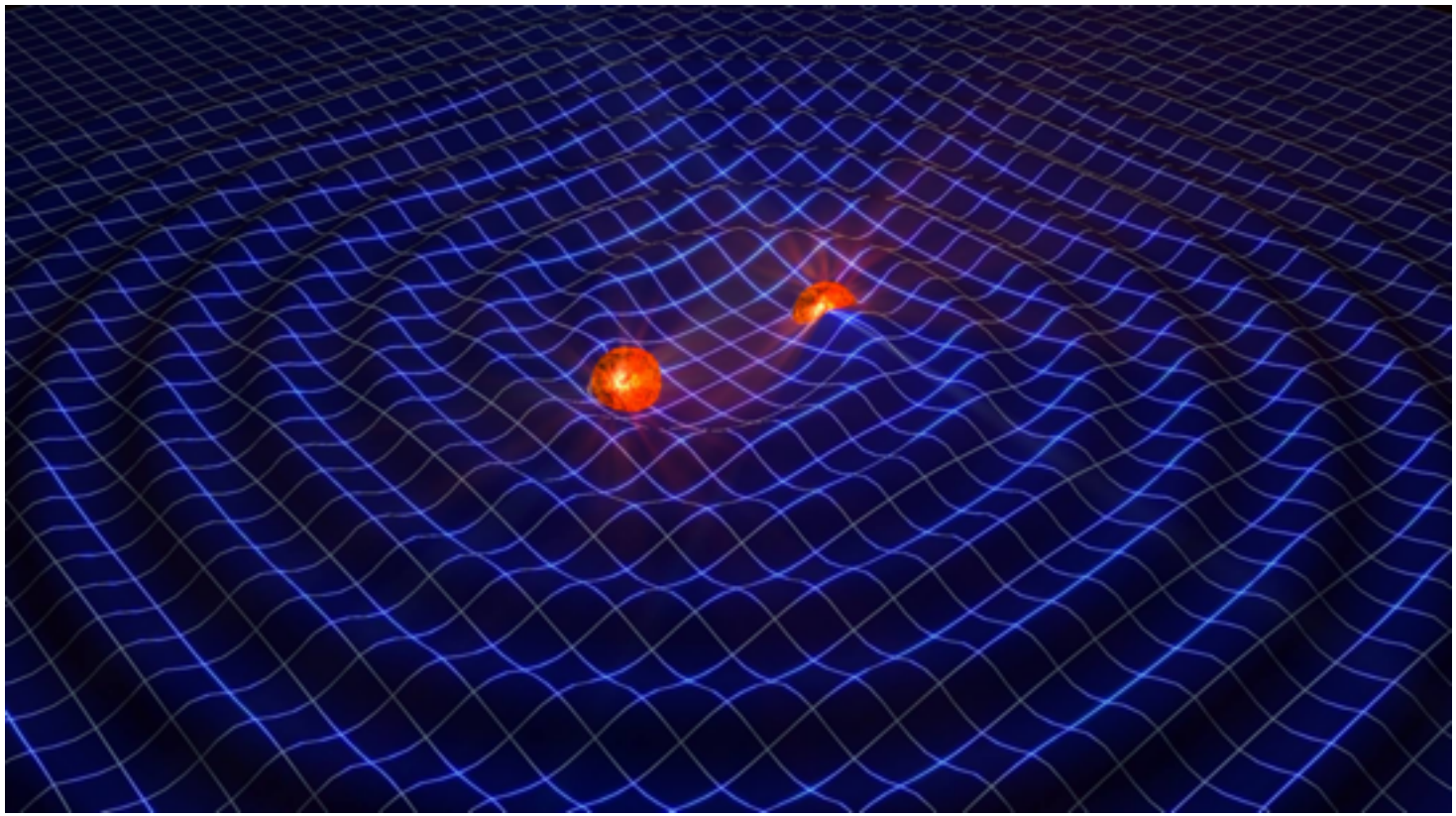
Type Casting

```
public class TypeCastDouble {

    public static void main(String[] args){

        double myDouble = 420.5;

        //Type cast double to int
        int i = (int)myDouble;

        System.out.println(i);
    }
}
```

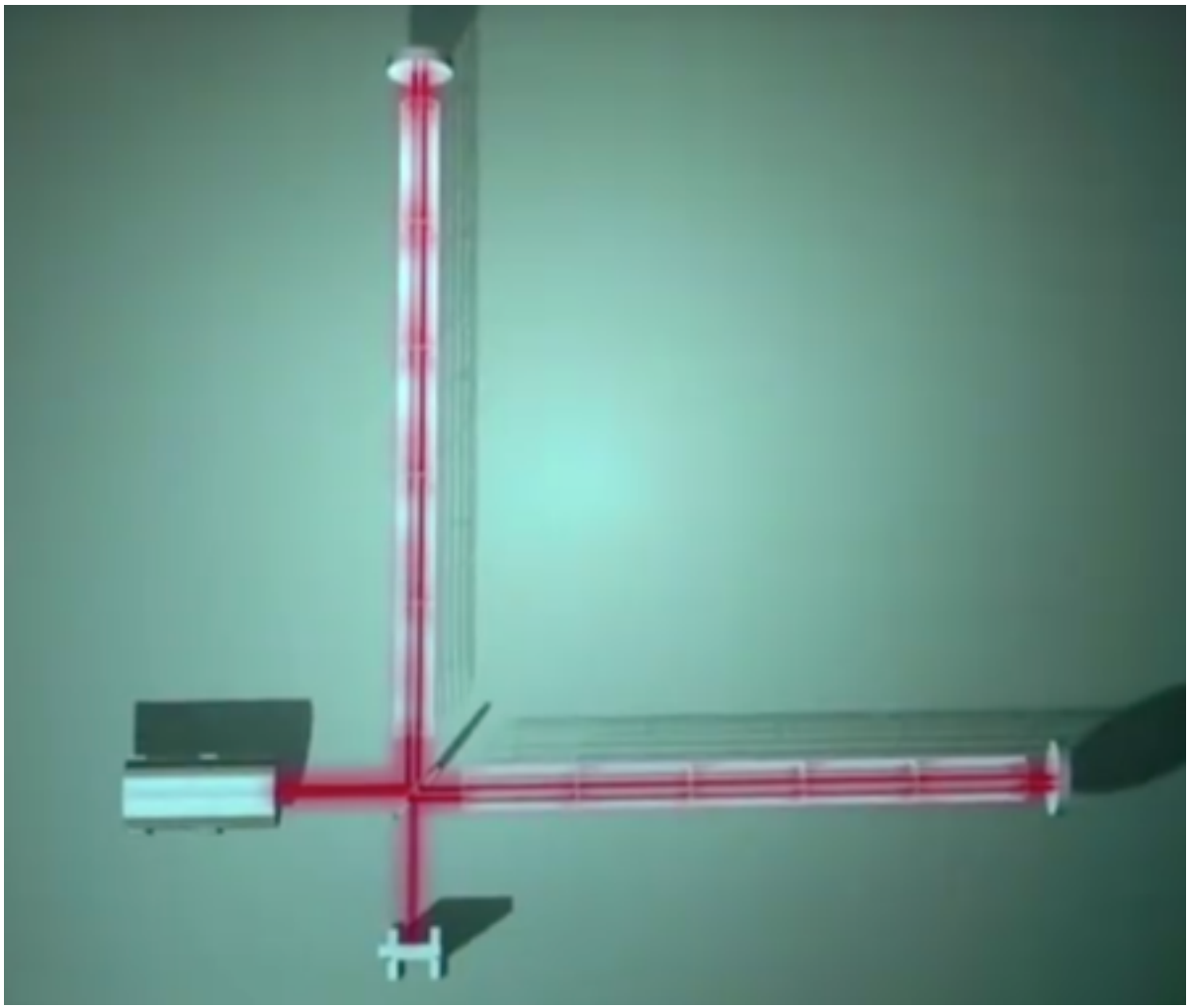
- Lets also take a look on TypeCasting.java and CatsAndDogs.java in code examples

Mental break

Gravitational waves



Mental break



Set interface

- A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains *only* methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

- Lets check the FindDups.java in code examples

Set Implementations

- The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet.
- HashSet, which stores its elements in a hash table, is the best-performing implementation, however it makes no guarantees concerning the order of iteration.
- TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.
- LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

Set Interface Bulk Operations (part 1)

- *s1.containsAll(s2)* — returns true if s2 is a **subset** of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
- *s1.addAll(s2)* — transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
- *s1.retainAll(s2)* — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
- *s1.removeAll(s2)* — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

Set Interface Bulk Operations (part 2)

- Union

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2); //obedinenie
```

- Intersection

```
Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2); //sechenie
```

- Difference

```
Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2); //razlika
```

- The implementation type of the result Set in the preceding idioms is HashSet, which is the best all-around Set implementation in the Java platform. However you can use any other general Set implementation.

List Interface

- A List is an ordered Collection (sometimes called a *sequence*).
- Lists may contain duplicate elements.
- In addition to the operations inherited from *Collection*, the *List* interface includes operations for the following:
  - Positional access — manipulates elements based on their numerical position in the list
  - Search — searches for a specified object in the list and returns its numerical position
  - Iteration — extends Iterator semantics to take advantage of the list's sequential nature
  - Range-view — performs arbitrary *range operations* on the list.

List Implementations

- The Java platform contains two general-purpose List implementations:
  - ArrayList, which is usually the better-performing implementation.
  - LinkedList which offers better performance under certain circumstances.

List - Collection Operations (part 1)

- The operations inherited from *Collection* all do about what you'd expect them to do, assuming you're already familiar with them.
- The *remove* operation always removes *the first* occurrence of the specified element from the list.
- The *add* and *addAll* operations always append the new element(s) to the *end* of the list.

List - Collection Operations (part 2)

- Thus, the following idiom concatenates one list to another:

```
list1.addAll(list2);
```

- Here's a *nondestructive* form of this idiom, which produces a third List consisting of the second list appended to the first.

```
List<Type> list3 = new ArrayList<Type>(list1);
list3.addAll(list2);
```

List - Positional Access and Search Operations

- The basic positional access operations (*get, set, add* and *remove*) behave just as you expect it.
- The search operations *indexOf* and *lastIndexOf* behave just as you expect it.
- The addAll operation inserts all the elements of the specified Collection starting at the specified position. The elements are inserted in the order they are returned by the specified Collection's iterator.

Hang on, equals?

Set Interface Bulk Operations (part 1)

- *Override equals*
- *Override hashcode*

Sorting Lists

- Comparable
- Comparator
- Collections.sort

Map Interface

- A Map is an object that maps keys to values:
  - A map cannot contain duplicate keys
  - Each key can map to at most one value.

Map – Iterate over it (part 1)

If you're only interested in the keys, you can iterate through the `keySet()` of the map:

```
Map<String, Object> map = ...;

for (String key : map.keySet()) {
    // ...
}
```

If you only need the values, use `values()`:

```
for (Object value : map.values()) {
    // ...
}
```

Finally, if you want both the key and value, use `entrySet()`:

```
for (Map.Entry<String, Object> entry : map.entrySet()) {
    String key = entry.getKey();
    Object value = entry.getValue();
    // ...
}
```

Map – Iterate over it (part 1)

- Lets take a look on MapExample.java in code examples

Summary

- What is array
- How to declare and initialize array
- How to access and change elements
- How to get the array length
- How to read an array from the console
- How to copy an array
- What is a matrix
- Collection, Iterator, For-each, List, Set, Map
- Sorting