A *type* is a set of values together with operations on them.

> Example: Type *integer*. The set of values is {…, -2, -1, 0, 1, 2, …}. The operations are +, -, *, / =, etc.
> Example: Java type **int**. The set of values is $\{-2^{31}..2^{31}-1\}$. The operations are +, - * / %, ==, <=, etc.

Data type is simply another, longer, name for *type*, although some people use it to refer mainly to types whose values are structured data, like sets, lists, trees, and graphs.

> Example: Java's *ArrayList* is a data type. The set of values are lists of elements. The operations are *size*(), *add*(*e*), *contains*(*e*), *get*(*i*), etc. The implementation is known: a list of values is maintained in a resizable array.

An *abstract data type*[1], or *ADT*, is a type whose implementation is not specified.

> Example: Type *set-of-integers*. The values are sets of integers, e.g. {3, 5, 2}. Its operations include (don't be concerned with the syntax of the operations; that is not the point here):
>
> | | |
> |---|---|
> | *size*(*S*) | —number of elements in set *S* |
> | *add*(S, x) | —add integer *x* to set *S* |
> | *remove*(*S*, *x*) | —remove *x* from set *S* |
> | *isIn*(*x*, S) | —return true iff integer *x* is in set *S* |
>
> Note that nothing is said about how type *set-of-integers* should be implemented.

Using a Java interface to define an abstract data type

An interface consists basically of a set of abstract methods. We can therefore think of an interface as defining an abstract data type, where the *syntax* of operations on the set of values is defined by the abstract methods and the set of values and meaning of these operations are given in comments.

For example, consider Java's interface *Set<E>*, whose specification in Java version 8 can be found here: docs.oracle.com/javase/8/docs/api/java/util/Set.html. A small part of its declaration is given in the text box at the bottom of the page. We gather from that declaration that:

1. The values in type *Set<E>* are the sets of elements of type *E* —collections of elements with no duplicates.
2. The operations on values of type *Set<E>* are (1) *add*(*E e*), (2) *contains*(*E e*), and *size*().
3. The meanings (semantics) of these operations are given in the Javadoc specs that precede the declarations of the operations.
4. More explanation of the set of values and the operations on them are given in the long Javadoc comment that precedes the class header.

```
/** A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2
  * such that e1.equals(e2) … As implied by its name, this interface models the mathematical set abstraction. … */
public interface Set<E> extends Collection<E> {

    /** Add e to this set if not already in it and return true iff it was added to the set. */
    public boolean add(E e);

    /** Return true if this set contains e. */
    public boolean contains(E e);

    /** Return the number of elements in this set. */
    public int size();

    …
}
```

[1] The term *abstract data type* appeared first in a paper by Barbara Liskov and Stephen Zilles in the Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, pp 50–59, 28-29 March 1974.

Making use of interfaces in a program

There are at least three implementations of Java interface *Set<E>*:

HashSet<E>,
LinkedHashSet<E>, and
TreeSet<E>, this one being useful only on types *E* that are comparable, like *Integer*.

Each implementation is better in a certain situation; it depends on which operations of interface *Set* will be executed most often and perhaps on how big a set will get.

Suppose we write a class *C* that needs a field that is a set of *Integer*s. We write the class with this field as shown to the right. Note: We have created a *HashSet*, but the type of the variable to which it is assigned is *Set*. That's an interface!

```
public class C {
   private Set<Integer> s= new HashSet<>());

   public void m() {
         …
         s.add(5);
         …
    }
}
```

This means that code in methods like *m* can use *only* methods that are defined in interface *Set*. That's OK! That's what we want. We expect to use only *Set* operations.

After programming for a while on this large class *C*, we realize that *HashSet* is not the best implementation to use. Because of the frequency of certain operations being performed on *s*, we believe that *TreeSet* is a better implementation in this environment.

To change to a *TreeSet*, all we have to do is change the declaration of this field:

private *Set<Integer> s*= **new** *TreeSet<>*());

To summarize: We declare *s* with the interface type for a reason. It ensures that all the code that uses *s* views *s* simply as a set. Then, later on, we can change the implementation simply by changing the initial assignment to *s*.