# Collections

Collections are Java classes developed to efficiently store multiple elements of a certain type. Unlike Java arrays collections can store any number of elements. The simplest collection is `ArrayList`, which you can treat as a resizable array. The following line of code creates an (initially empty) `ArrayList` of objects of class `Person`:

```
ArrayList<Person> friends = new ArrayList<Person>();
```

Note that the type of the collection includes the element type in angle brackets. This "tunes" the collection to work with the specific element type, so that, for example, the function `get()` of `friends` will return object of type `Person`. The `ArrayList<Person>` offers the following API (for the complete API see Java Class Reference):

- `int size()` – returns the number of elements in the list
- `boolean isEmpty()` – tests if this list has no elements
- `Person get( int index )` – returns the element at the specified position in this list
- `boolean add( Person p )` – appends the specified element to the end of this list
- `Person remove( int index )` – removes the element at the specified position in this list
- `boolean contains( Person p )` – returns true if this list contains a given element
- `void clear()` – removes all of the elements from this list

The following code fragment tests if the `friends` list contains the person `victor` and, if, victor is not in there, adds him to the list:

```
if( ! friends.contains( victor ) )
    friends.add( victor );
```

All collection types support iteration over elements. The simplest construct for iteration is a "for" loop. Suppose the class `Person` has a field `income`. The loop below prints all friends with income greater than 100000 to the model log:

```
for( Person p : friends ) {
    if( p.income > 100000 )
        traceln( p );
}
```

Another popular collection type is `LinkedList`.

Linked lists are used to model stack or queue structures, i.e. sequential storages where elements are primarily added and removed from one or both ends.

Consider a model of a distributor that maintains a backlog of orders from retailers. Suppose there is an `Order` class with the `amount` field. The backlog (essentially a FIFO queue) can be modeled as:

```
LinkedList<Order> backlog = new LinkedList<Order>();
```

`LinkedList` supports functions common to all collections (like `size()` or `isEmpty()`) and also offers a specific API:

- `Order getFirst()` – returns the first element in this list

- `Order getLast()` – returns the last element in this list
- `addFirst( Order o )` – inserts the given element at the beginning of this list
- `addLast( Order o )` – appends the given element to the end of this list
- `Order removeFirst()` – removes and returns the first element from this list
- `Order removeLast()` – removes and returns the last element from this list

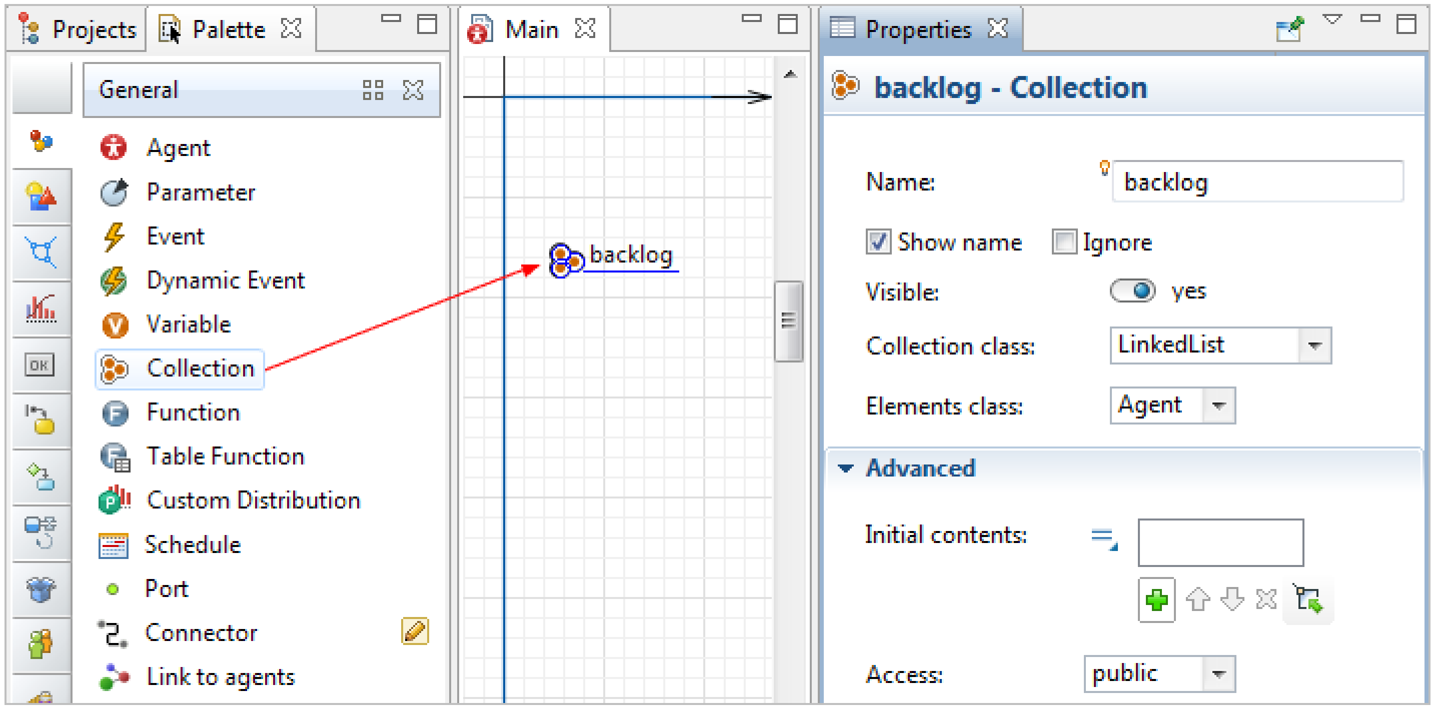
When a new order is received by the distributor it is placed at the end of the backlog:

```
backlog.addLast( order );
```

Each time the inventory gets replenished, the distributor tries to ship the orders starting from the head of the backlog. If the amount in an order is bigger than the remaining inventory, the order processing stops. The order processing can look like this:

```
while( ! backlog.isEmpty() ) { //repeat the code below while the backlog is not empty
    Order order = backlog.getFirst(); //pick the first order in the backlog
    if( order.amount <= inventory ) { //if enough inventory to satisfy this order
        ship( order ); //ship
        inventory -= order.amount; //decrease available inventory
        backlog.removeFirst(); //remove the order from the backlog
    } else { //not enough inventory to ship
        break; //stop order backlog processing
    }
}
```

We recommend to declare collections in agents and experiments graphically. The **Collection** object is located in the **Agent** palette. All you need to do is to drop it on the canvas and choose the collection and the element types. At runtime you will be able to view the collection contents by clicking its icon.



backlog - Collection

Name: backlog

[✓] Show name  [ ] Ignore

Visible: yes

Collection class: LinkedList

Elements class: Agent

Initial contents:

Access: public

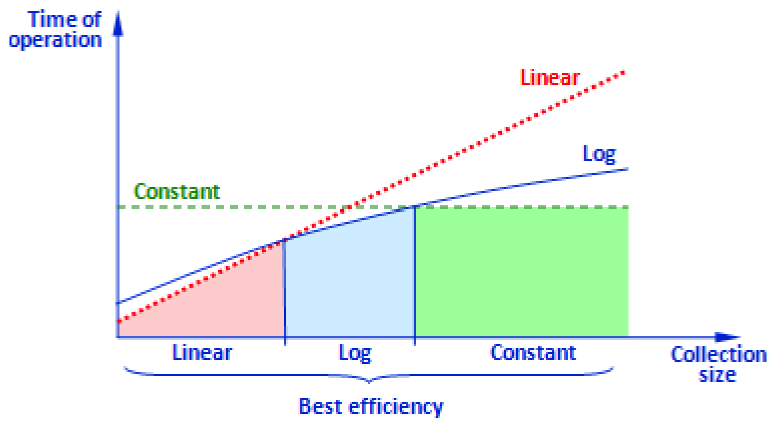
*Declaring Java collection graphically*

Different types of collections have different *time complexity* of operations. For example, checking if a collection of 10,000,000 objects contains a given object may take 80 milliseconds for `ArrayList`, 100 milliseconds for `LinkedList`, and less than 1 millisecond for `HashSet` and `TreeSet`. To ensure maximum efficiency of the model execution you should analyze, which operations are most frequent and choose the collection type correspondingly. The Table below contains the time complexities of the most common operations for four collection types.

| Operation | ArrayList | LinkedList | HashSet | TreeSet |
|---|---|---|---|---|
| Obtain size | Constant | Constant | Constant | Constant |
| Add element | Constant | Constant | Constant | Log |
| Remove given element | Linear | Linear | Constant | Log |
| Remove by index | Linear | Linear | – | – |
| Get element by index | Constant | Linear | – | – |
| Find out if contains | Linear | Linear | Constant | Log |

The terms *constant*, *linear*, and *logarithmic complexity* mean the following. Linear complexity means that the worst time required to complete the operation grows linearly as the size of the collection grows. Constant means that it does not depend on the size at all, and Log means the time grows as the logarithm of the size. You should not treat the constant complexity as the unconditionally best choice. Depending on the size, different types of collection may behave better than others.

Consider the Figure. It may well happen that with relatively small number of elements the collection with linear complexity will behave better than the one with constant complexity.



*Depending on the size, different types of collections may behave better that others*

And a couple of things you should keep in mind when choosing the collection type:

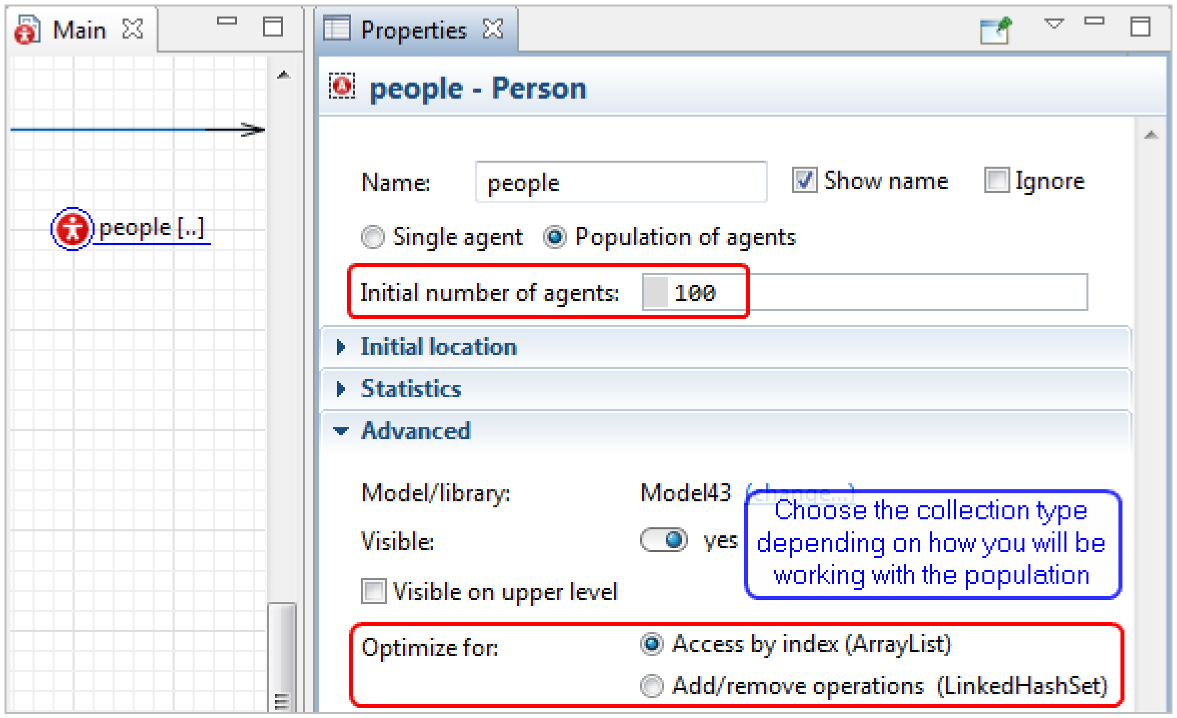
- `HashSet` and `TreeSet` do not support element indexes, so it is not possible to e.g. get an element at position 32.
- `TreeSet` is a naturally sorted collection: elements are stored in a certain order defined by a natural or a user-provided comparator.

### Agent populations are collections too

When you declare an embedded agent as replicated, AnyLogic creates a special type of collection to store the individual agents. You have two options:

- `AgentArrayList` – choose this collection type if the set of agents is more or less constant or if you need to frequently access individual objects by index.
- `AgentLinkedHashSet` – choose this option if you plan to intensively add new agents and remove existing ones. For example, if you are modeling a population of a city for a relatively long period so that people are born, die, move out of the city, and new people arrive.

The options for the type of collection appear in the **Advanced** section of the embedded object properties, see the Figure.



**people - Person**

Name: people [✓ Show name] Ignore

Single agent [● Population of agents]

Initial number of agents: 100

Initial location

Statistics

Advanced

Model/library: Model43 (change...)

Visible: yes

Visible on upper level

Choose the collection type depending on how you will be working with the population

Optimize for: [● Access by index (ArrayList)] Add/remove operations (LinkedHashSet)

*Options for collection type of a replicated object*

Both collections support functions like `size()`, `isEmpty()`, `get( int index )`, `contains( Object ao )`, and iteration. If index is not specifically needed during iteration, it is always better to use the enhanced form of "for" loop:

```
for( Person p : people ) {
    …
}
```