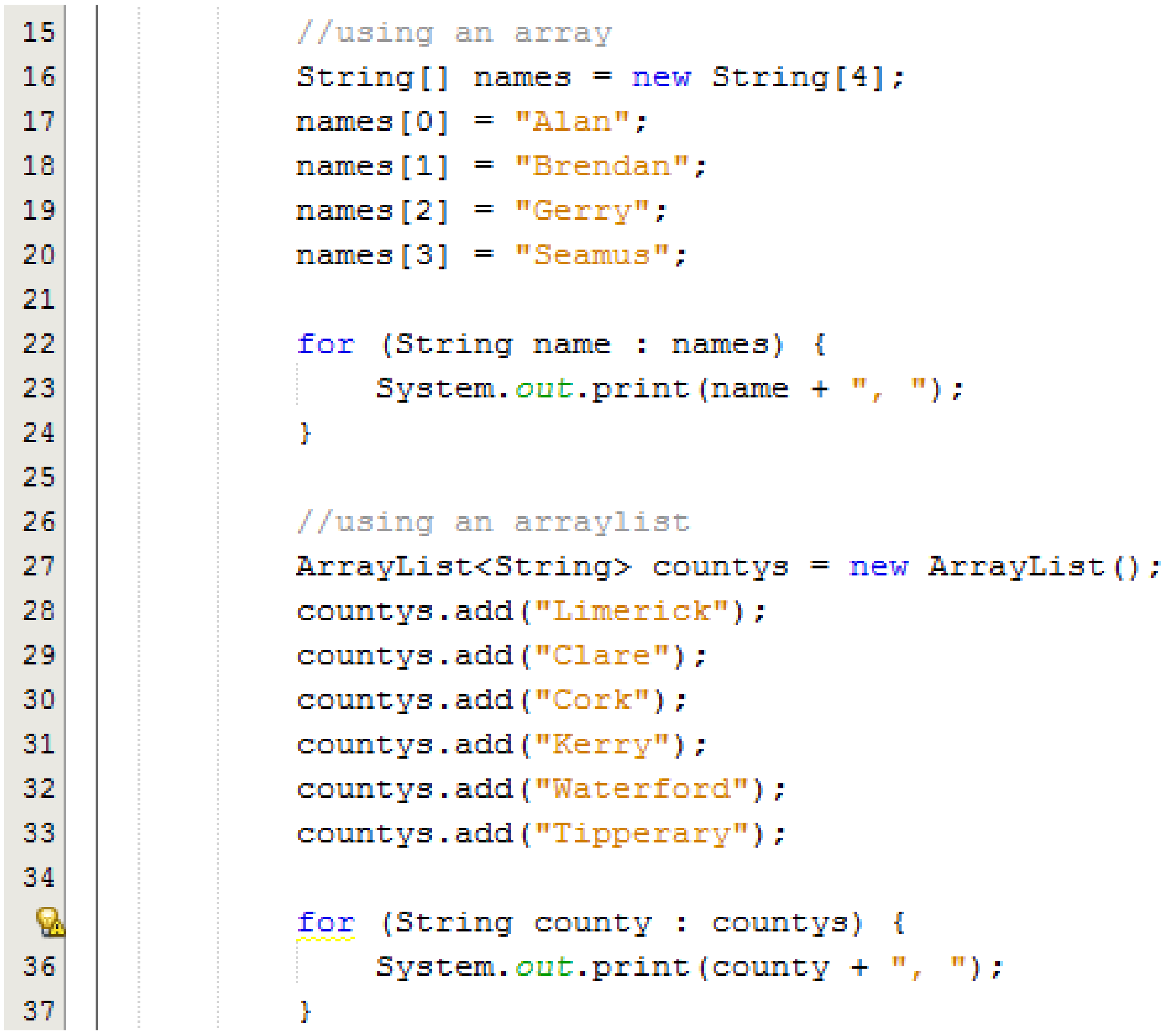
B.Sc. In Software Development. Year 3.
Applications Programming.
Collections, Generics and Lambdas.

LIMERICK INSTITUTE OF TECHNOLOGY
SCHOOL OF SCIENCE, ENGINEERING & I.T.
Department of Information Technology

Introduction

- Like an array, a collection is an object that can hold one or more elements.
- However, unlike arrays, collections aren’t part of the language itself.
- Instead, collections are classes that are available from the Java API.

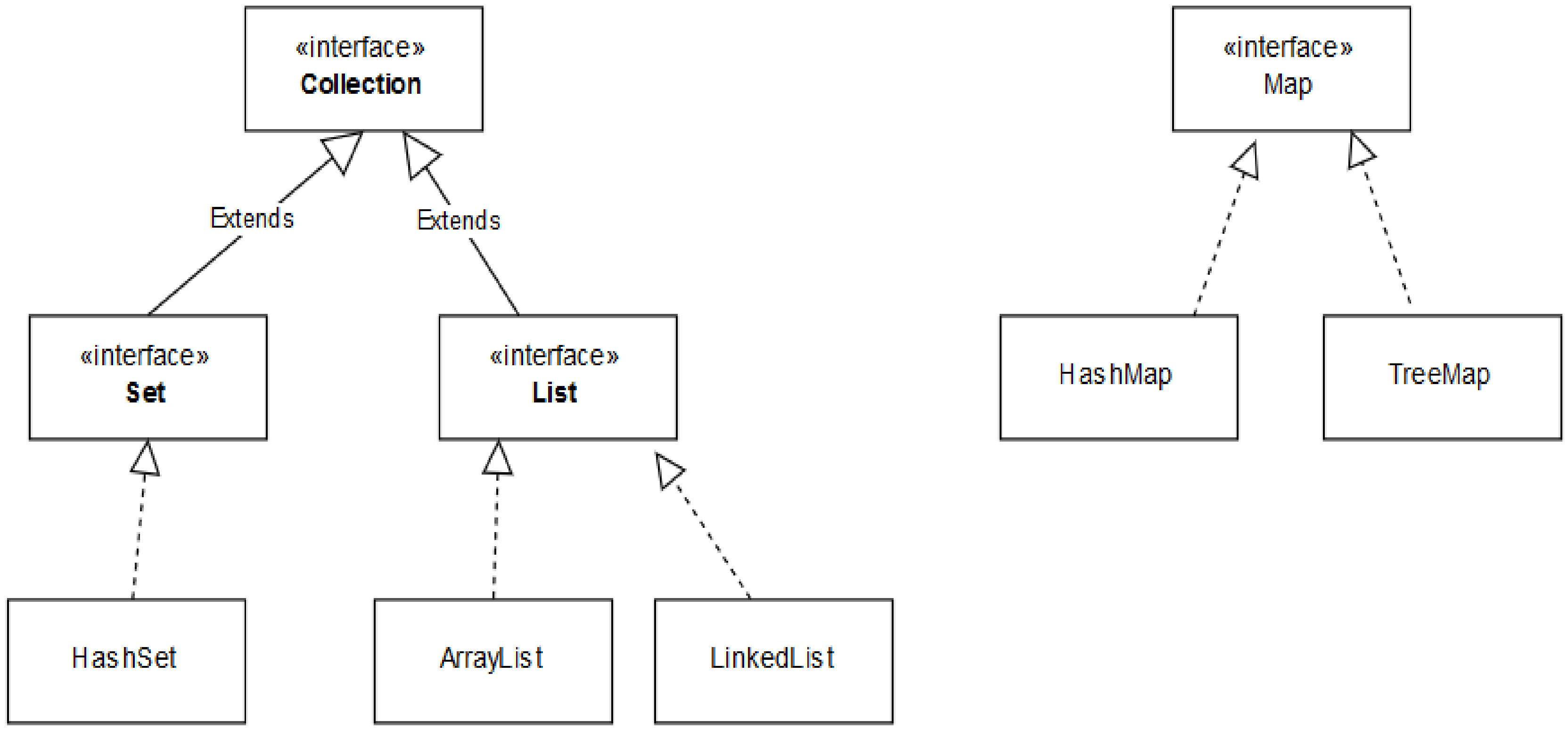
Introduction



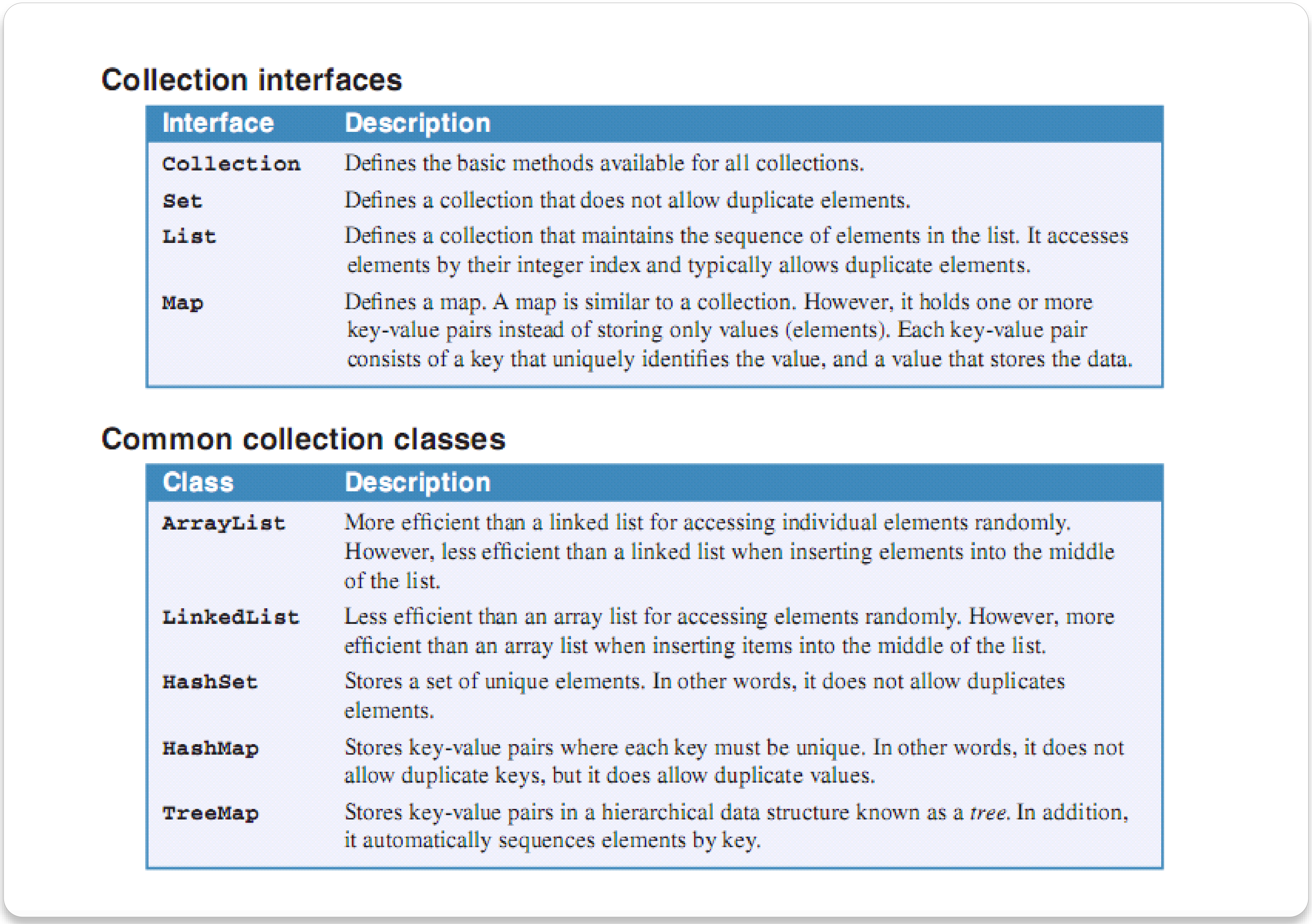
```
//using an array
String[] names = new String[4];
names[0] = "Alan";
names[1] = "Brendan";
names[2] = "Gerry";
names[3] = "Seamus";

for (String name : names) {
    System.out.print(name + ", ");
}

//using an arraylist
ArrayList<String> countys = new ArrayList();
countys.add("Limerick");
countys.add("Clare");
countys.add("Cork");
countys.add("Kerry");
countys.add("Waterford");
countys.add("Tipperary");

for (String county : countys) {
    System.out.print(county + ", ");
}
```

Overview of the Collection Framework

- The framework consists of a hierarchy of interfaces and classes.



Overview of the Collection Framework



Collection interfaces

Interface	Description
`Collection`	Defines the basic methods available for all collections.
`Set`	Defines a collection that does not allow duplicate elements.
`List`	Defines a collection that maintains the sequence of elements in the list. It accesses elements by their integer index and typically allows duplicate elements.
`Map`	Defines a map. A map is similar to a collection. However, it holds one or more key-value pairs instead of storing only values (elements). Each key-value pair consists of a key that uniquely identifies the value, and a value that stores the data.

Common collection classes

Class	Description
`ArrayList`	More efficient than a linked list for accessing individual elements randomly. However, less efficient than a linked list when inserting elements into the middle of the list.
`LinkedList`	Less efficient than an array list for accessing elements randomly. However, more efficient than an array list when inserting items into the middle of the list.
`HashSet`	Stores a set of unique elements. In other words, it does not allow duplicates elements.
`HashMap`	Stores key-value pairs where each key must be unique. In other words, it does not allow duplicate keys, but it does allow duplicate values.
`TreeMap`	Stores key-value pairs in a hierarchical data structure known as a *tree*. In addition, it automatically sequences elements by key.

Generics and Collections

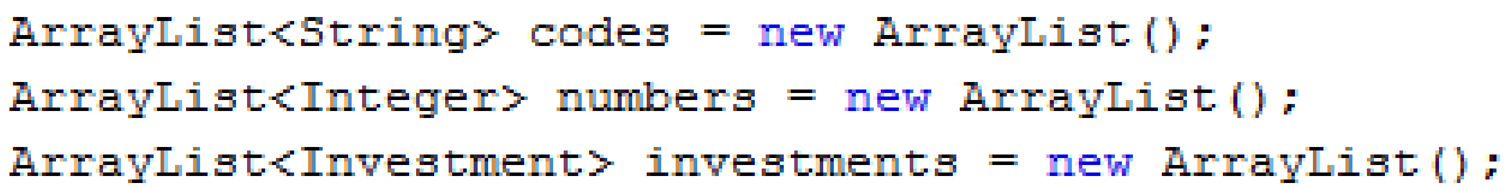
- Prior to Java 5, the elements of a collection were defined as the Object type. At first glance, this might seem like an advantage.
- However, there are two (glaring) disadvantages:
- Java 5 introduced generics tha0t addresses these two problems.
    - Let you specify the element type for a collection.
- Java will then ensure that it only adds objects of the specified type to the collection.
- Conversely, Java can automatically cast any objects you retrieve from the collection to the correct type.

Generics and Collections

- However, there are two (glaring) disadvantages:
- How to specify elements in a collection:

```
CollectionClass<Type> name =
                                             new CollectionClass();
```

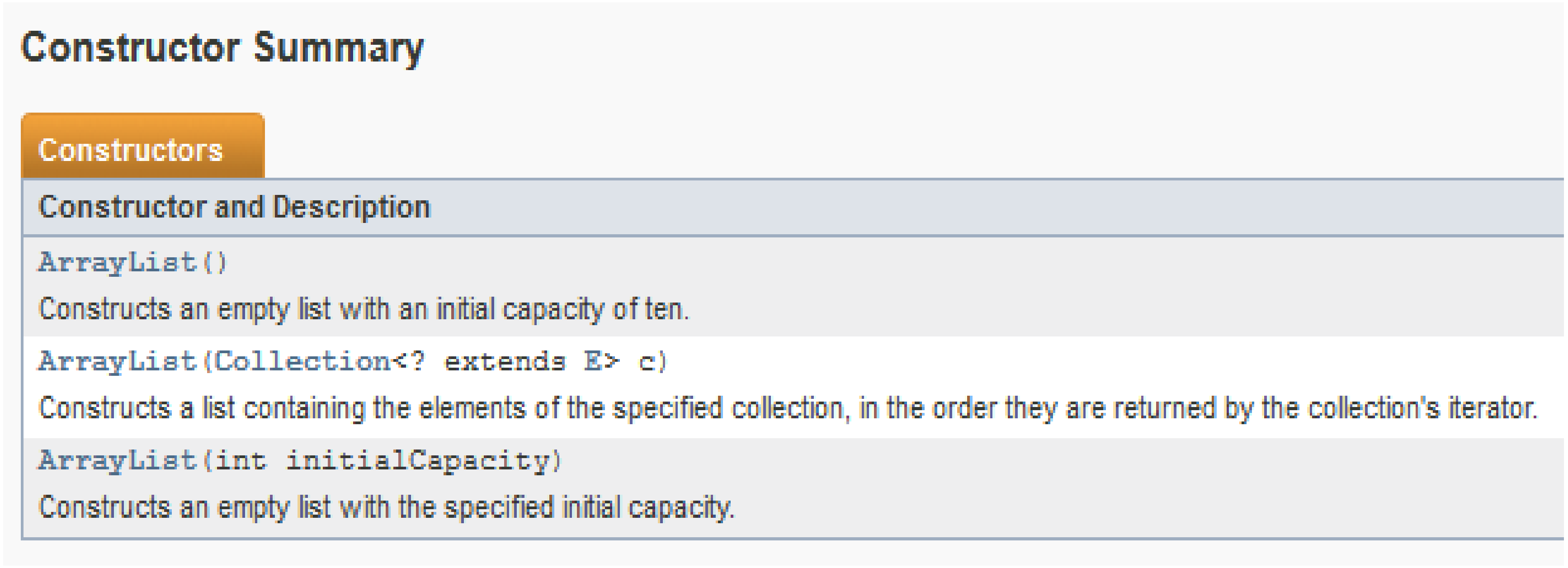
- Examples:



```
ArrayList<String> codes = new ArrayList();
ArrayList<Integer> numbers = new ArrayList();
ArrayList<Investment> investments = new ArrayList();
```

How To Use ArrayLists

- ArrayLists are one of the most commonly used collections.
- Are similar to arrays, however, their size automatically adjusts its size as you add elements to it.



Constructor Summary

Constructors

Constructor and Description
`ArrayList()` Constructs an empty list with an initial capacity of ten.
`ArrayList(Collection<? extends E> c)` Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
`ArrayList(int initialCapacity)` Constructs an empty list with the specified initial capacity.

How To Use ArrayLists

Common Methods Of the ArrayList Class	
add(object)	Adds the specified object to the end of the list
add(index, object)	Adds the specified object at the specified index position
get(index)	Returns the object at the specified index
size()	Returns the number of elements in the list
clear()	Removes all the elements from the list
contains(object)	Returns true if the specified object is in the list

How To Use ArrayLists

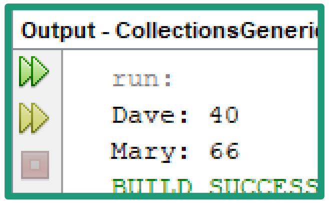
Common Methods Of the ArrayList Class	
isEmpty()	Returns true if the list is empty
remove(index)	Removes the object at the specified index and returns that object
remove(object)	Removes the specified object and returns a Boolean that indicates whether the operation was successful.
set(index, object)	Updates the element at the specified index to the specified object
toArray()	Returns an array containing the elements of the list

Generic Classes

- You often need to implement classes and methods that work with multiple types.
- E.G. `ArrayList<T>` stores elements of an arbitrary class `T`.
  - The `ArrayList` is deemed to be generic.
  - T is a type parameter.

Generic Classes

```java
public class Entry <K,V>{

    private K key;
    private V value;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}//end class
```



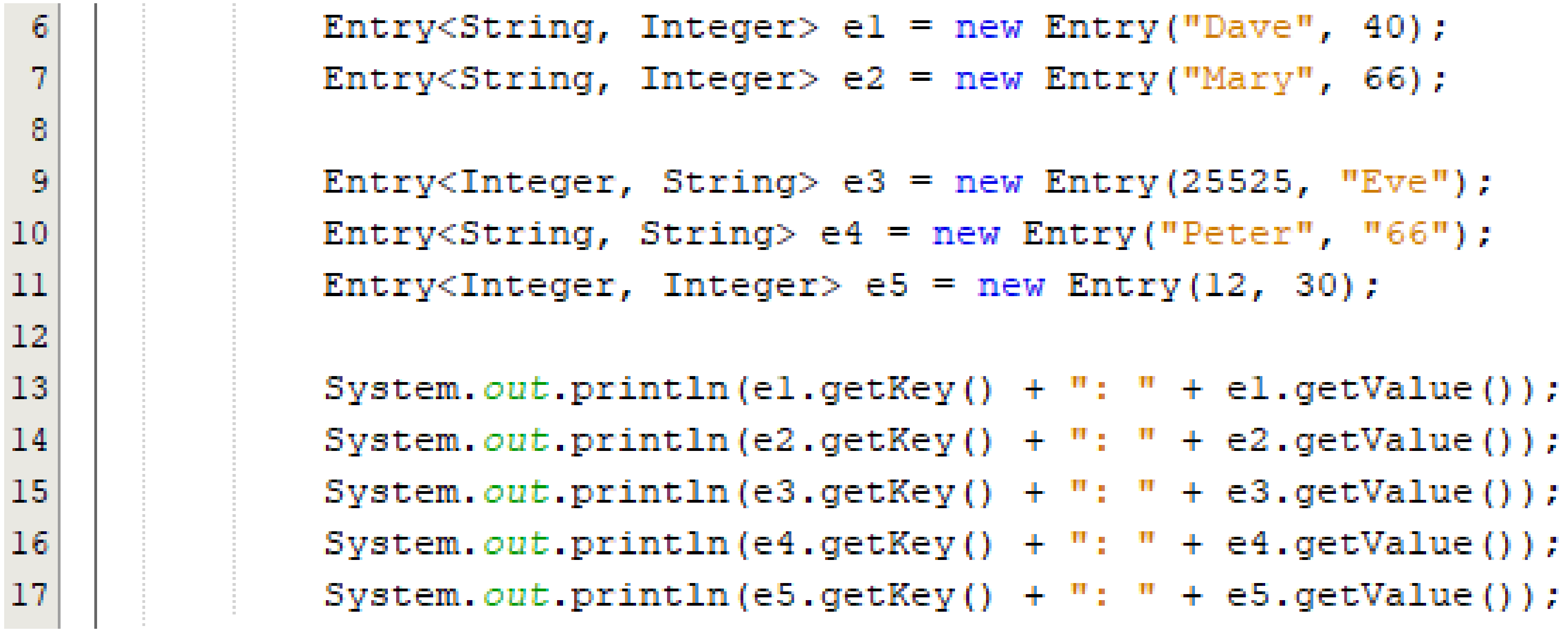
Output - CollectionsGeneri

```
run:
Dave: 40
Mary: 66
BUILD SUCCESS
```

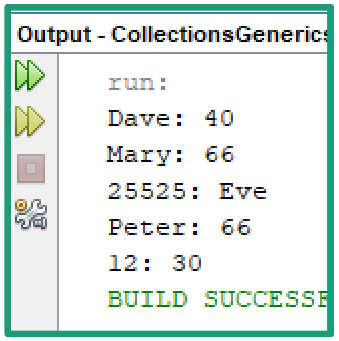
```java
    Entry<String, Integer> anEntry = new Entry("Dave", 40);
    Entry<String, Integer> anotherEntry = new Entry("Mary", 66);

    System.out.println(anEntry.getKey() + ": " + anEntry.getValue());
    System.out.println(anotherEntry.getKey() + ": " + anotherEntry.getValue());
```

Generic Classes

- Generic classes can be used in a variety of ways.



```java
        Entry<String, Integer> e1 = new Entry("Dave", 40);
        Entry<String, Integer> e2 = new Entry("Mary", 66);

        Entry<Integer, String> e3 = new Entry(25525, "Eve");
        Entry<String, String> e4 = new Entry("Peter", "66");
        Entry<Integer, Integer> e5 = new Entry(12, 30);

        System.out.println(e1.getKey() + ": " + e1.getValue());
        System.out.println(e2.getKey() + ": " + e2.getValue());
        System.out.println(e3.getKey() + ": " + e3.getValue());
        System.out.println(e4.getKey() + ": " + e4.getValue());
        System.out.println(e5.getKey() + ": " + e5.getValue());
```

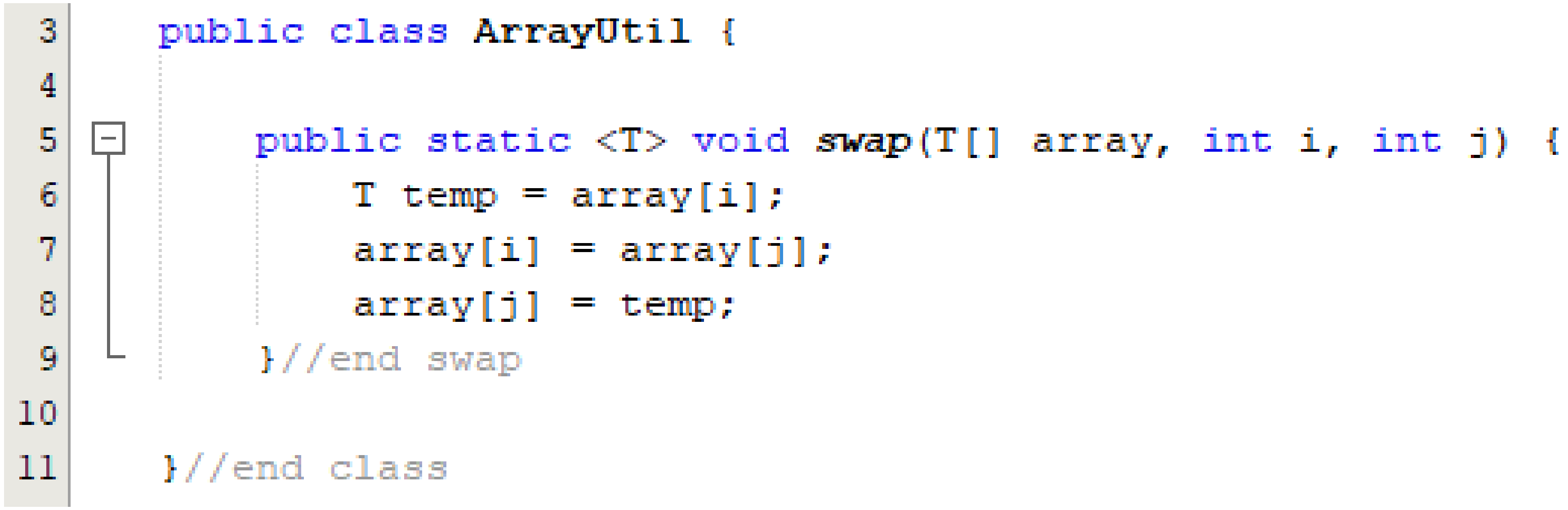


```
run:
Dave: 40
Mary: 66
25525: Eve
Peter: 66
12: 30
BUILD SUCCESSF
```

generics.TestEntry.java

Generic Methods

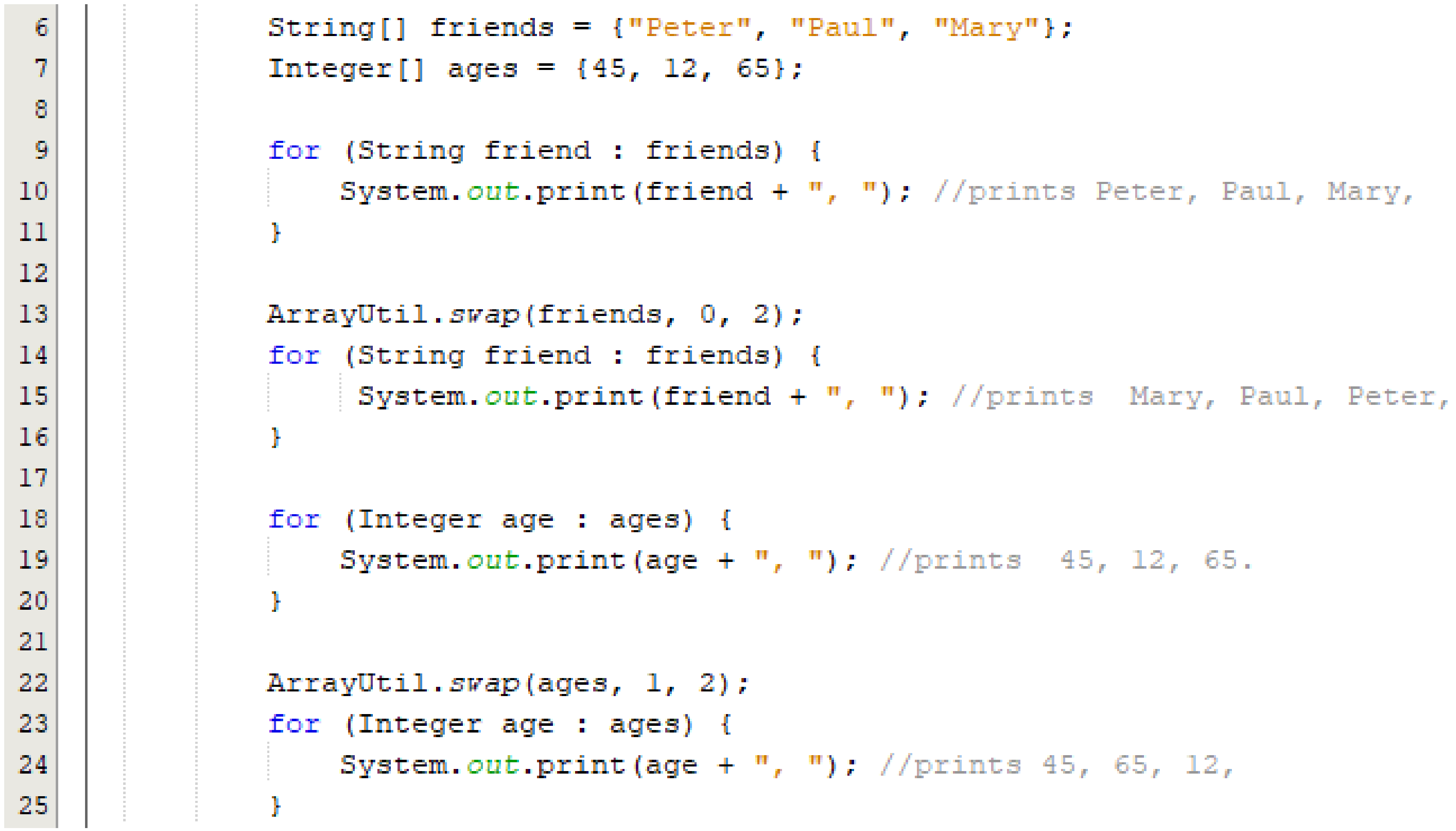
- A method with type parameters.
- Can appear in Generic and non Generic classes.



```java
public class ArrayUtil {

    public static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }//end swap

}//end class
```

generics. ArrayUtil.java

Calling Generic Methods

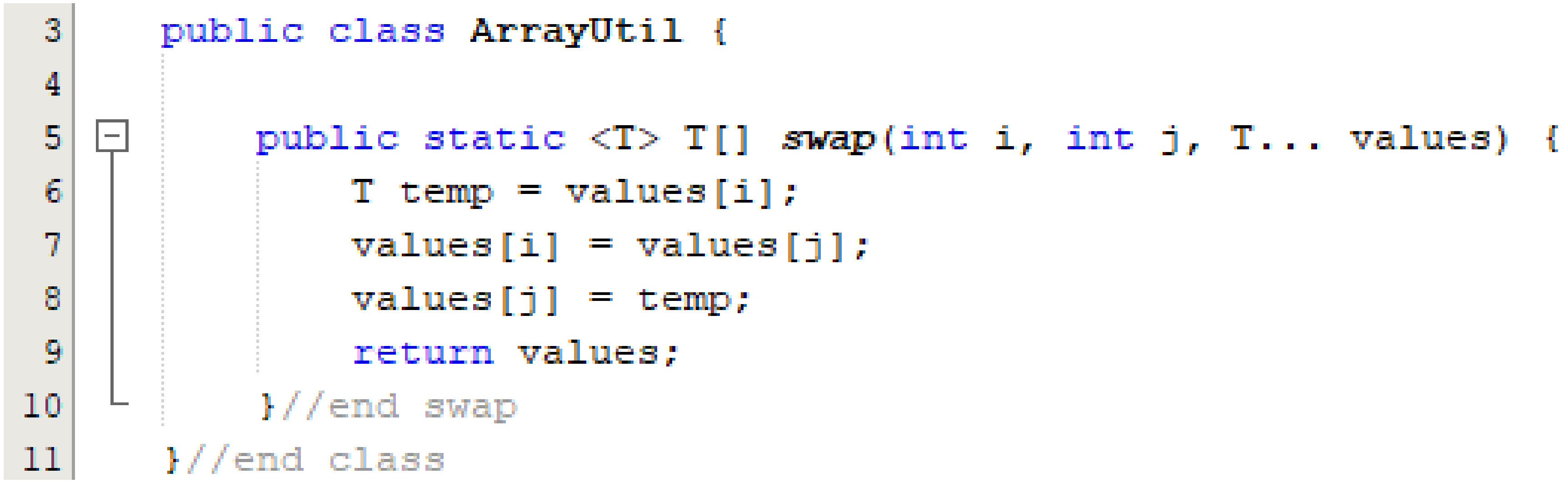


```java
String[] friends = {"Peter", "Paul", "Mary"};
Integer[] ages = {45, 12, 65};

for (String friend : friends) {
    System.out.print(friend + ", "); //prints Peter, Paul, Mary,
}

ArrayUtil.swap(friends, 0, 2);
for (String friend : friends) {
     System.out.print(friend + ", "); //prints  Mary, Paul, Peter,
}

for (Integer age : ages) {
    System.out.print(age + ", "); //prints  45, 12, 65.
}

ArrayUtil.swap(ages, 1, 2);
for (Integer age : ages) {
    System.out.print(age + ", "); //prints 45, 65, 12,
}
```

generics.ArrayUtilDemo.java

Generic Methods

- Consider this overloaded version of swap

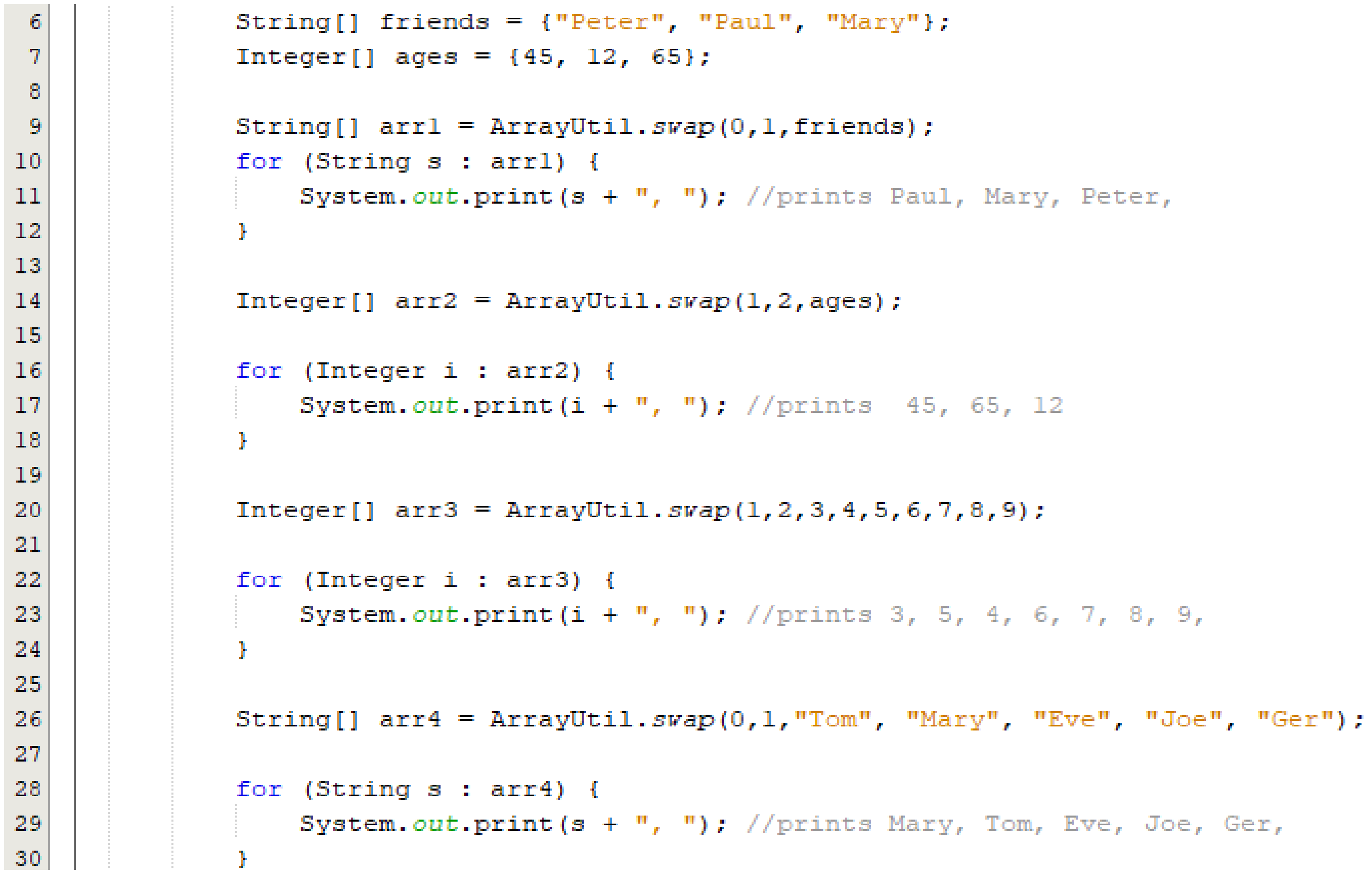


```
public class ArrayUtil {

    public static <T> T[] swap(int i, int j, T... values) {
        T temp = values[i];
        values[i] = values[j];
        values[j] = temp;
        return values;
    }//end swap
}//end class
```

- How would you call it?

generics. ArrayUtil.java

Calling Generic Methods



```java
String[] friends = {"Peter", "Paul", "Mary"};
Integer[] ages = {45, 12, 65};

String[] arr1 = ArrayUtil.swap(0,1,friends);
for (String s : arr1) {
    System.out.print(s + ", "); //prints Paul, Mary, Peter,
}

Integer[] arr2 = ArrayUtil.swap(1,2,ages);

for (Integer i : arr2) {
    System.out.print(i + ", "); //prints  45, 65, 12
}

Integer[] arr3 = ArrayUtil.swap(1,2,3,4,5,6,7,8,9);

for (Integer i : arr3) {
    System.out.print(i + ", "); //prints 3, 5, 4, 6, 7, 8, 9,
}

String[] arr4 = ArrayUtil.swap(0,1,"Tom", "Mary", "Eve", "Joe", "Ger");

for (String s : arr4) {
    System.out.print(s + ", "); //prints Mary, Tom, Eve, Joe, Ger,
}
```

generics.ArrayUtilDemo.java

Lambda Expressions

- A block of code that you can pass around so that it can be executed later, once or multiple times.
- Are arguably the most important new feature of Java 8.
- They are similar in some ways to a method in an anonymous class.
- Allow you to pass the functionality of a method as a parameter.
- Sometimes called anonymous functions.

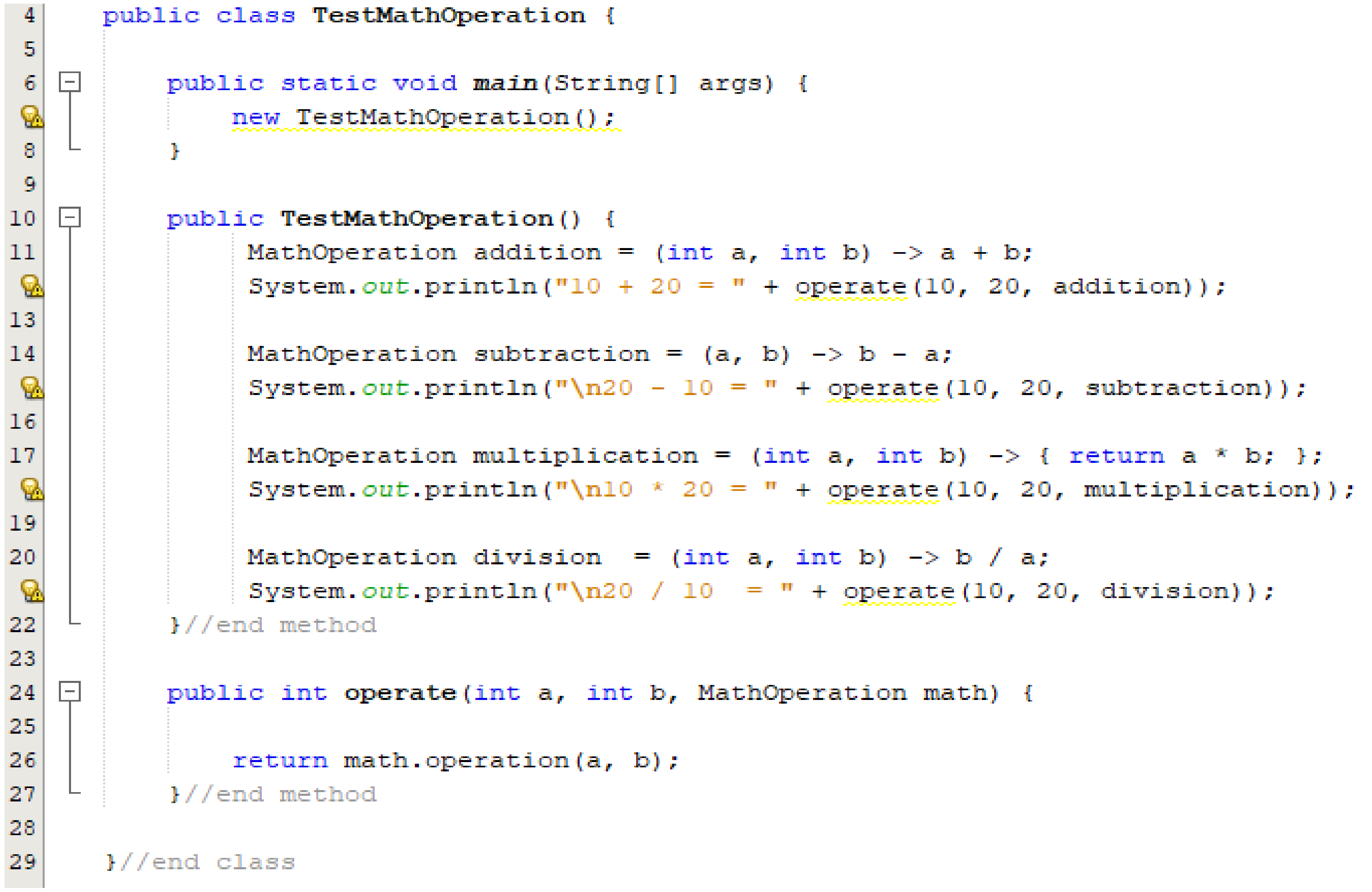
Lambda Expressions

- Lambda expressions have a cleaner syntax than anonymous class.
- They allow you to store functionality of a method and pass it to another method as a parameter.
- The ability to treat functionality if it were data can result in following benefits.
- Can reduce code duplication.
- Allow you to write methods that are more flexible and easier to maintain.

Lambda Expressions

- There are also drawbacks that may mean you do not always use them.
- Lambda expressions can be difficult to debug because you can’t step through them with the debugger like standard methods.
- When a lambda throws an exception, the stack trace can be difficult to understand.
- Methods that use lambdas can sometimes be inefficient compared to methods that accomplish the same task without using them.
- Can result in code that’s difficult to read/maintain.

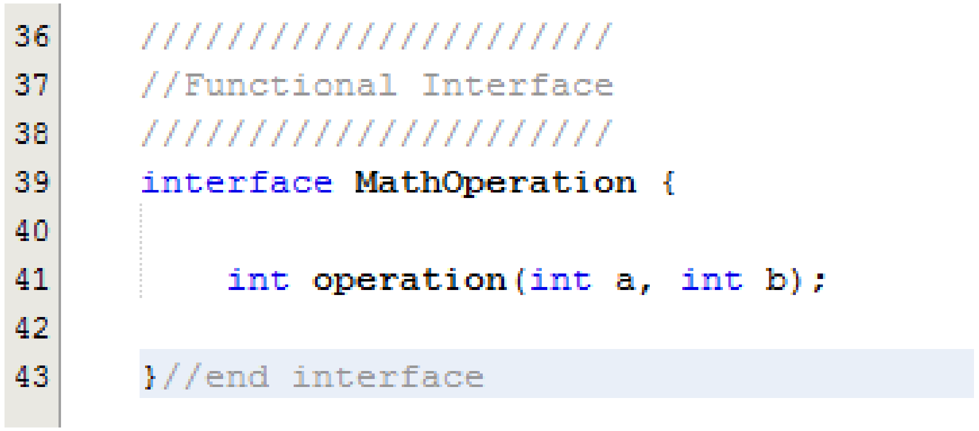
Lambda Example



```java
public class TestMathOperation {

    public static void main(String[] args) {
        new TestMathOperation();
    }

    public TestMathOperation() {
        MathOperation addition = (int a, int b) -> a + b;
        System.out.println("10 + 20 = " + operate(10, 20, addition));

        MathOperation subtraction = (a, b) -> b - a;
        System.out.println("\n20 - 10 = " + operate(10, 20, subtraction));

        MathOperation multiplication = (int a, int b) -> { return a * b; };
        System.out.println("\n10 * 20 = " + operate(10, 20, multiplication));

        MathOperation division  = (int a, int b) -> b / a;
        System.out.println("\n20 / 10  = " + operate(10, 20, division));
    }//end method

    public int operate(int a, int b, MathOperation math) {

        return math.operation(a, b);
    }//end method

}//end class
```

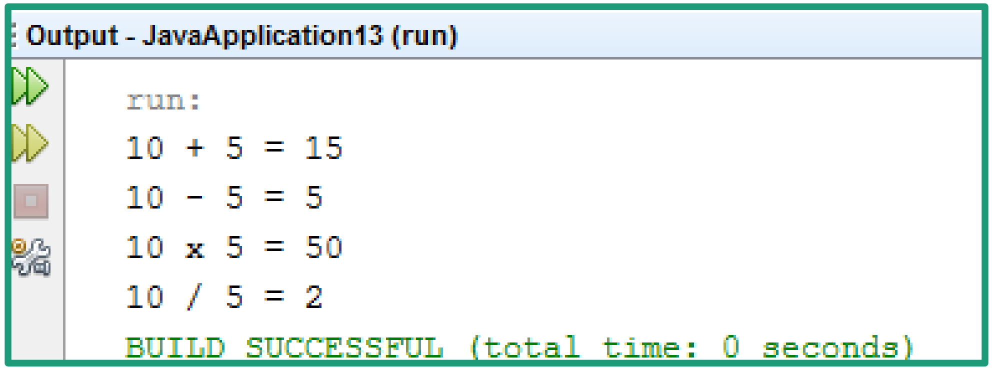
lambdas.basic.TestMathOperation.java

Lambda Example



```
///////////////////////
//Functional Interface
///////////////////////
interface MathOperation {

    int operation(int a, int b);

}//end interface
```

lambdas.basic.MathOperation.java

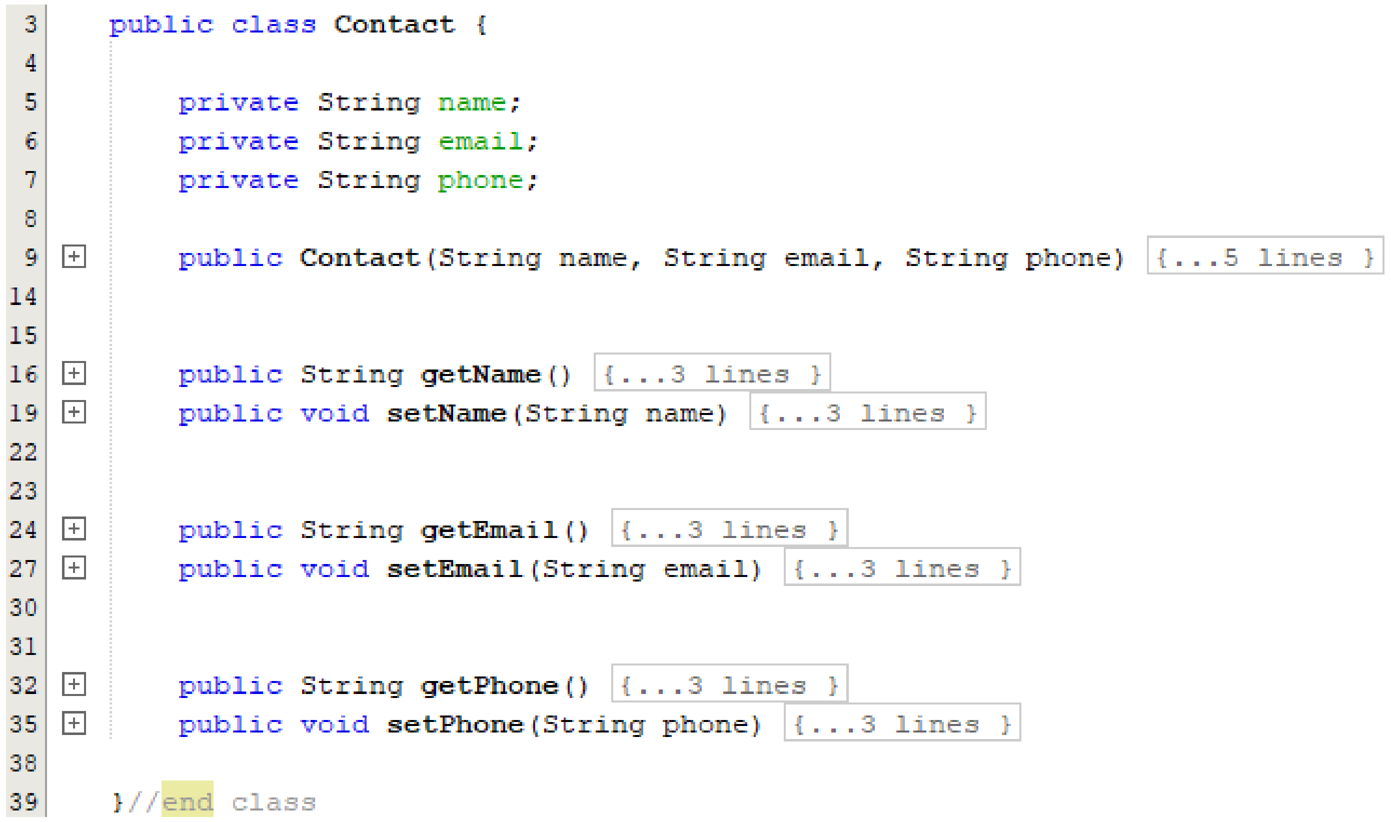


Output - JavaApplication13 (run)

```
run:
10 + 5 = 15
10 - 5 = 5
10 x 5 = 50
10 / 5 = 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

A method that doesn’t use a lambda expression

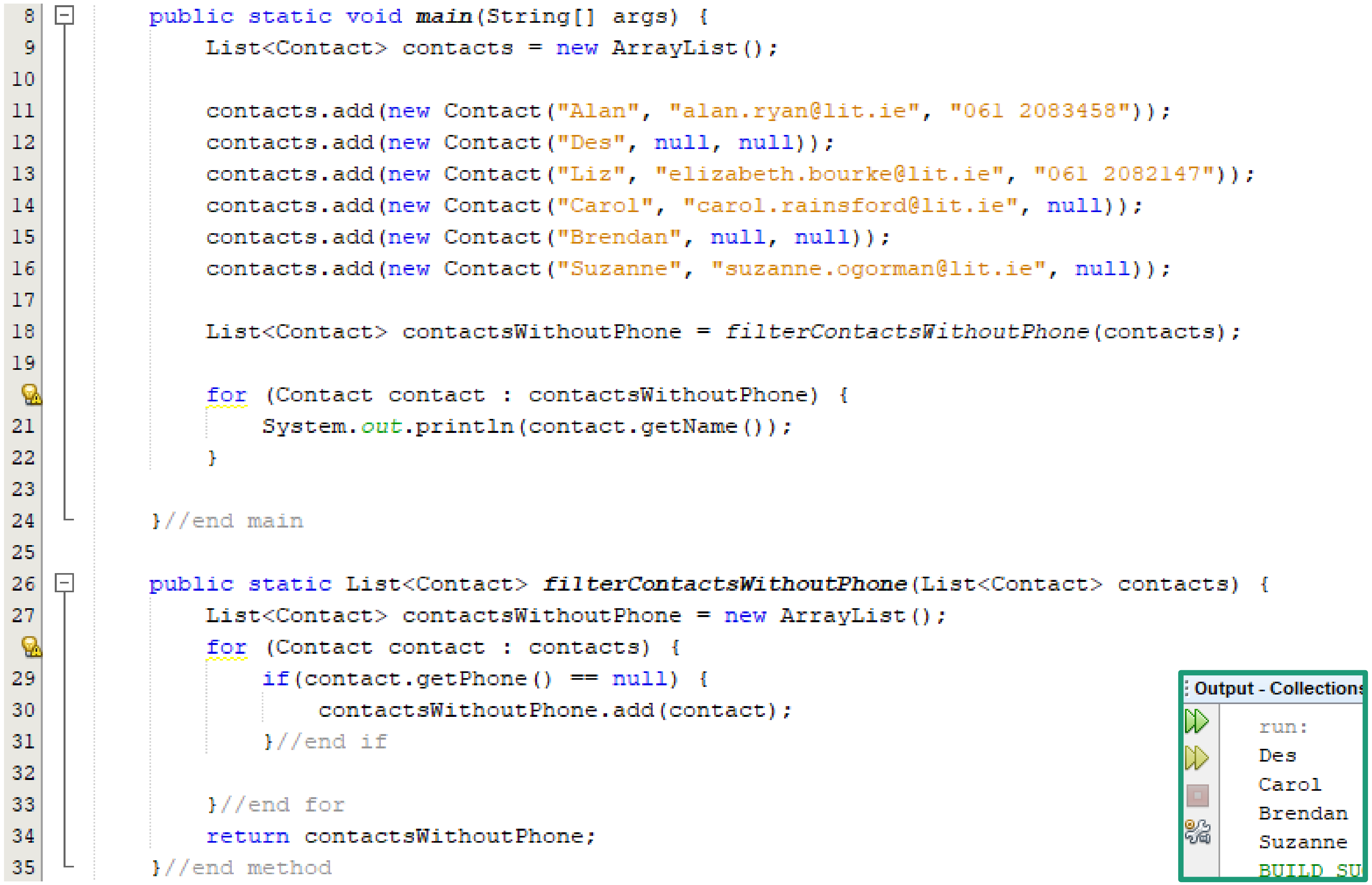
- The following code shows a method (next slide) that doesn’t use a lambda expression but that could benefit from using one.



```
public class Contact {

    private String name;
    private String email;
    private String phone;

    public Contact(String name, String email, String phone) {...5 lines }


    public String getName() {...3 lines }
    public void setName(String name) {...3 lines }


    public String getEmail() {...3 lines }
    public void setEmail(String email) {...3 lines }


    public String getPhone() {...3 lines }
    public void setPhone(String phone) {...3 lines }

}//end class
```

lambdas.advanced.Contact.java

A method that doesn't use a lambda expression



```java
    public static void main(String[] args) {
        List<Contact> contacts = new ArrayList();

        contacts.add(new Contact("Alan", "alan.ryan@lit.ie", "061 2083458"));
        contacts.add(new Contact("Des", null, null));
        contacts.add(new Contact("Liz", "elizabeth.bourke@lit.ie", "061 2082147"));
        contacts.add(new Contact("Carol", "carol.rainsford@lit.ie", null));
        contacts.add(new Contact("Brendan", null, null));
        contacts.add(new Contact("Suzanne", "suzanne.ogorman@lit.ie", null));

        List<Contact> contactsWithoutPhone = filterContactsWithoutPhone(contacts);

        for (Contact contact : contactsWithoutPhone) {
            System.out.println(contact.getName());
        }

    }//end main

    public static List<Contact> filterContactsWithoutPhone(List<Contact> contacts) {
        List<Contact> contactsWithoutPhone = new ArrayList();
        for (Contact contact : contacts) {
            if(contact.getPhone() == null) {
                contactsWithoutPhone.add(contact);
            }//end if

        }//end for
        return contactsWithoutPhone;
    }//end method
```

Output - Collections

```
run:
Des
Carol
Brendan
Suzanne
BUILD SU
```

lambdas.advanced.TestLambdas.java

A method that doesn’t use a lambda expression

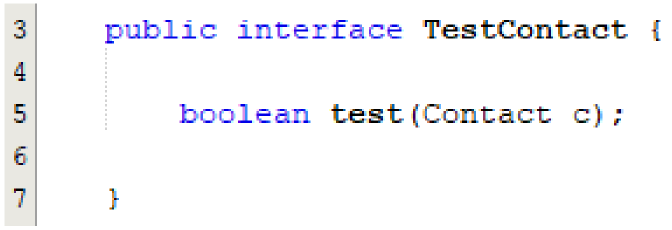
- Code duplication can become a problem.
  - If a change had to be made to the Contact class it, further changes are required in any filter methods.
  - In this situation it makes sense to use a lambda expression because it can make the method more flexible.
  - Increases maintainability, decreases code duplication.

A method that does use a lambda expression

- The following code shows how to perform the same task as the previous one with a method that uses a lambda.
- This results in a flexible method that you can use to filter the list of Contact objects in multiple ways.

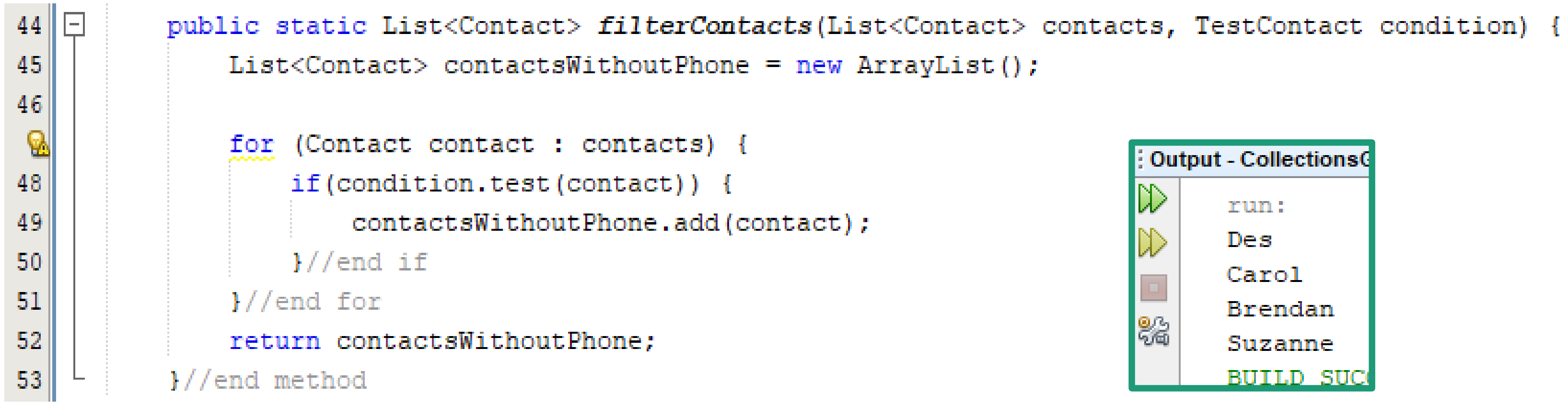
A method that does use a lambda expression

- First of all, a functional interface is needed.

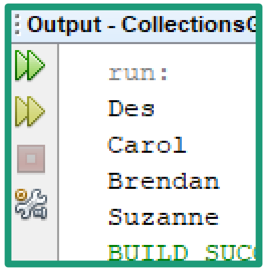


```java
public interface TestContact {

    boolean test(Contact c);

}
```

- A method that uses a functional interface to specify the filter condition.



```java
public static List<Contact> filterContacts(List<Contact> contacts, TestContact condition) {
    List<Contact> contactsWithoutPhone = new ArrayList();

    for (Contact contact : contacts) {
        if(condition.test(contact)) {
            contactsWithoutPhone.add(contact);
        }//end if
    }//end for
    return contactsWithoutPhone;
}//end method
```

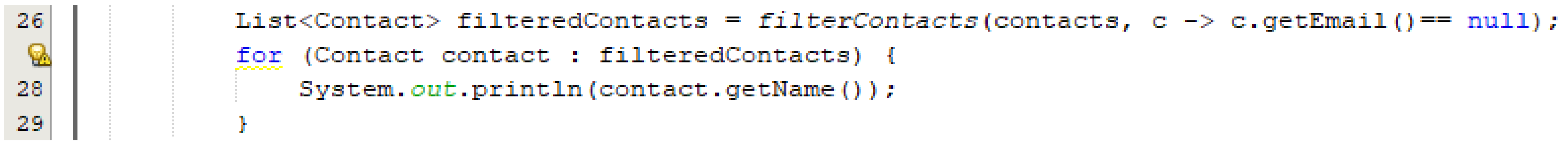


Output - CollectionsG

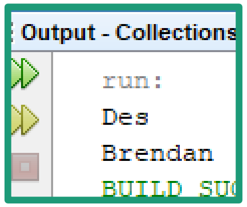
```
run:
Des
Carol
Brendan
Suzanne
BUILD SUC
```

lambdas.advanced.TestLambdas.java

A method that does use a lambda expression

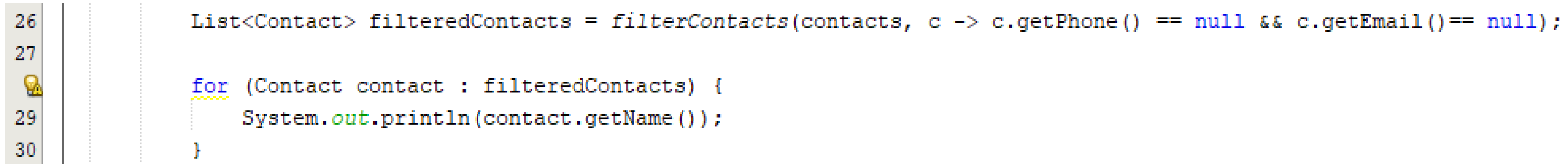


```
List<Contact> filteredContacts = filterContacts(contacts, c -> c.getEmail()== null);
for (Contact contact : filteredContacts) {
    System.out.println(contact.getName());
}
```

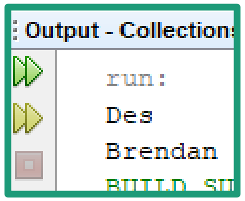


```
Output - Collections
run:
Des
Brendan
BUILD SU
```

- If you want you could code more complex lambda expressions to filter the list in other ways.



```
List<Contact> filteredContacts = filterContacts(contacts, c -> c.getPhone() == null && c.getEmail()== null);

for (Contact contact : filteredContacts) {
    System.out.println(contact.getName());
}
```



```
Output - Collection
run:
Des
Brendan
BUILD SU
```

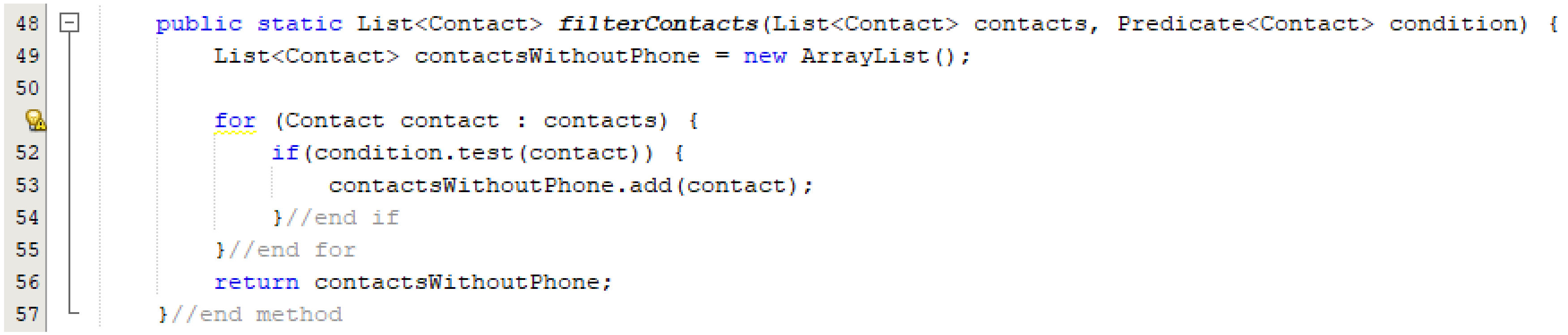
lambdas.advanced.TestLambdas.java

A method that does use a lambda expression

- Although this figure presents a simple example of using Lambdas, it should be clear to you how they can make methods more **flexible**, **reduce code duplication** and make code **easier to maintain**.
- You can replace multiple methods that perform almost identical tasks with a single method that allows the functionality to be passed in at runtime as a lambda expression.

Using The Predicate Interface

- The following shows how to perform the same task as the previous one with a method that uses the Predicate interface.
- This interface defines a method named test that works much like the test method in the TestContact functional interface from the previous example.



```
    public static List<Contact> filterContacts(List<Contact> contacts, Predicate<Contact> condition) {
        List<Contact> contactsWithoutPhone = new ArrayList();

        for (Contact contact : contacts) {
            if(condition.test(contact)) {
                contactsWithoutPhone.add(contact);
            }//end if
        }//end for
        return contactsWithoutPhone;
    }//end method
```

Using The Predicate Interface

- However, the Predicate interface has two advantages over the TestContact interface.
  - Firstly, its already available from the Java API. As a result you don’t need to write the code to define the interface.
  - Secondly, the Predicate interface uses generics to specify the type of object that’s passed to its test method.
- By contrast, the test method of the TestContact interface can only accept a Contact object.
- In this code, the second parameter of the filterContacts method defines a parameter of the Predicate<Contact> type.
- As a result, the lambda expressions that are passed to this method can call methods of the Contact object.

Future Reading

- Type Bounds.
- Type Variance and Wildcards.
- Higher-Order Functions.

References

Murach J., and Urban M. (2014) *Murach's Beginning Java with NetBeans*, Mike Murach and Associates, Inc. (Link)

Cay S. Horstmann (2018) Core Java SE 9 For the Impatient. 2/E. ISBN-13 978-0-13-469472-6 (Link)

https://www.tutorialspoint.com/java8/java8_lambda_expressions.htm

https://docs.oracle.com/javase/tutorial/java/generics/

https://www.tutorialspoint.com/java/java_generics.htm

http://tutorials.jenkov.com/java-generics/index.html

http://tutorials.jenkov.com/java/lambda-expressions.html