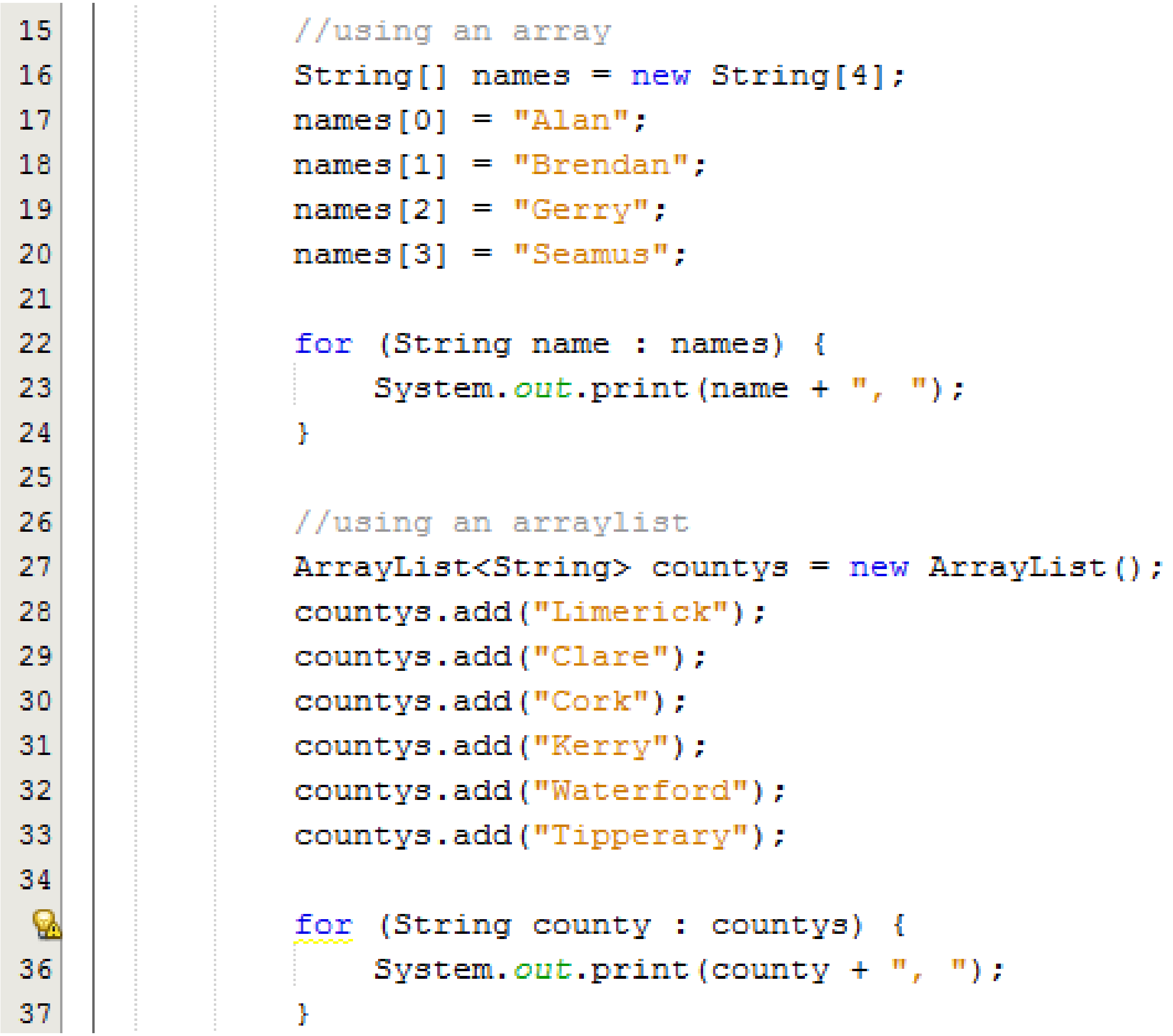
B.Sc. In Software Development. Year 3.
Applications Programming.
Collections, Generics and Lambdas

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.

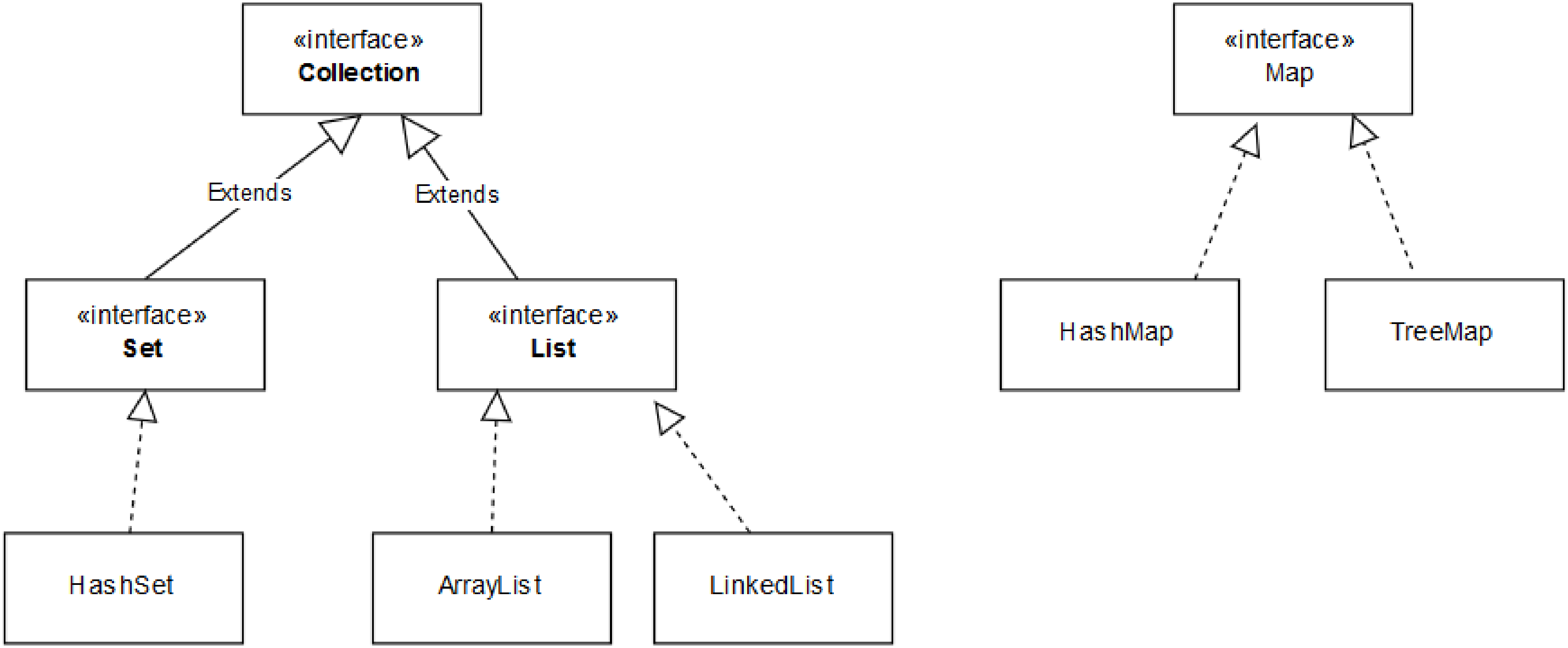
Introduction



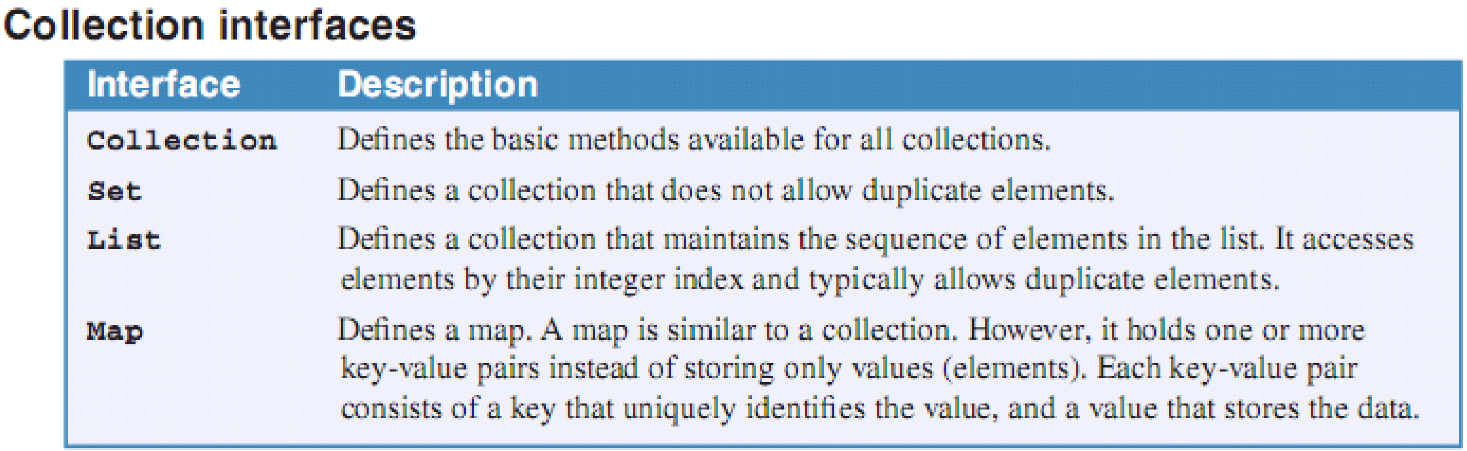
```java
//using an array
String[] names = new String[4];
names[0] = "Alan";
names[1] = "Brendan";
names[2] = "Gerry";
names[3] = "Seamus";

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.

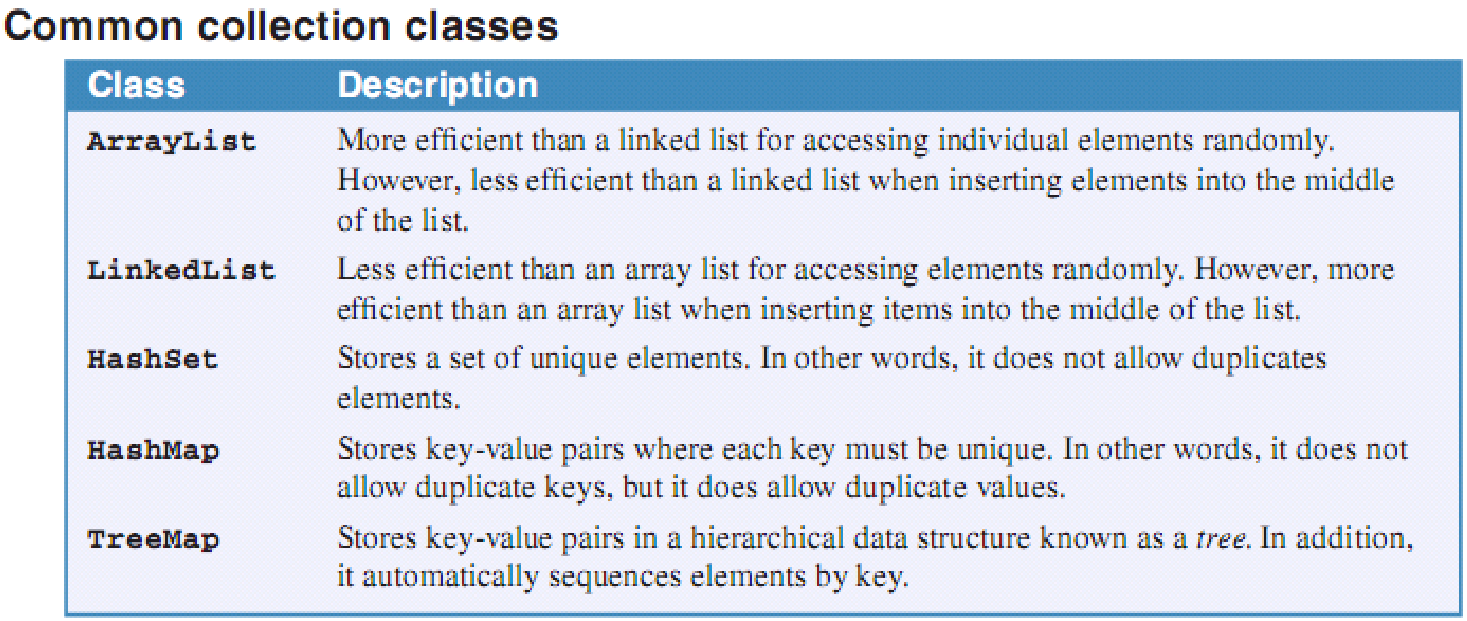


Overview of the Collection Framework



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.

Introduction to Generics

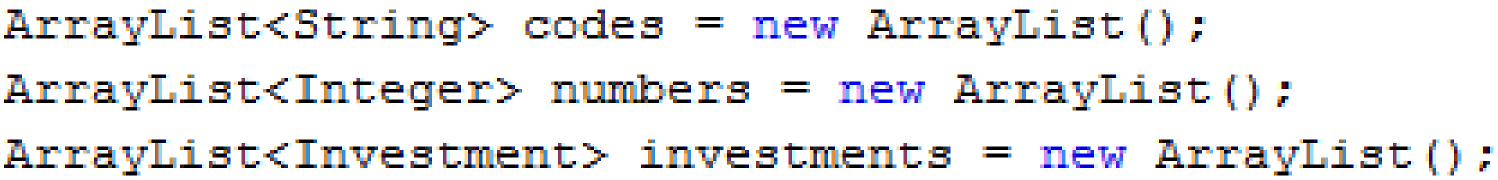
- Prior to Java 5, the elements of a collection were defined as the Object type. As a result, you could store any type of object as an element in a collection.
- At first glance, this might seem like an advantage. However, there are two (glaring) disadvantages:
- Java 5 introduced generics that addresses these two problems.
- Generics lets 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.

Introduction to Generics

- How to specify elements in a collection:

  CollectionClass<Type> name = new CollectionClass():

- Examples:



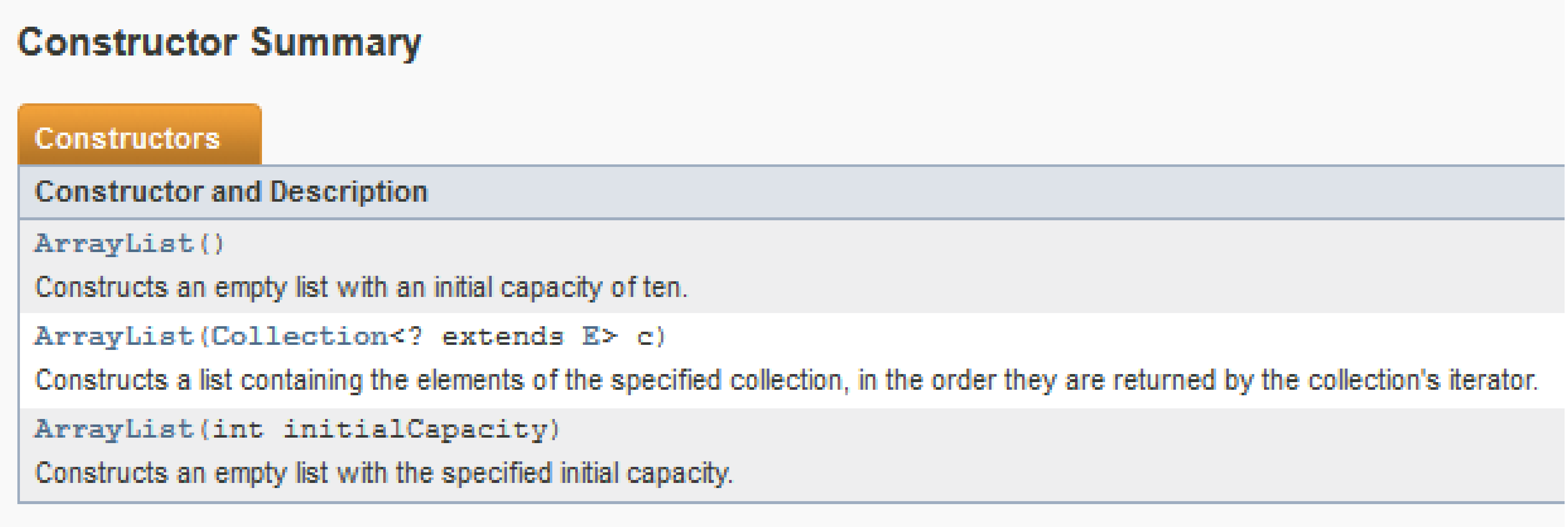
```
ArrayList<String> codes = new ArrayList();
ArrayList<Integer> numbers = new ArrayList();
ArrayList<Investment> investments = new ArrayList();
```

***Note 1:** its illegal to store primitive types directly in a collection.*

***Note 2:** if you omit a type for a collection it can store any type of object.*

How to use ArrayLists

- The ArrayList class is one of the most commonly used collections.
- ArrayLists as mentioned 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.

Using 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
indexOf(object)	Returns the index position of the specified object

Using 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

Using Lambdas

- Lambda expressions 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.

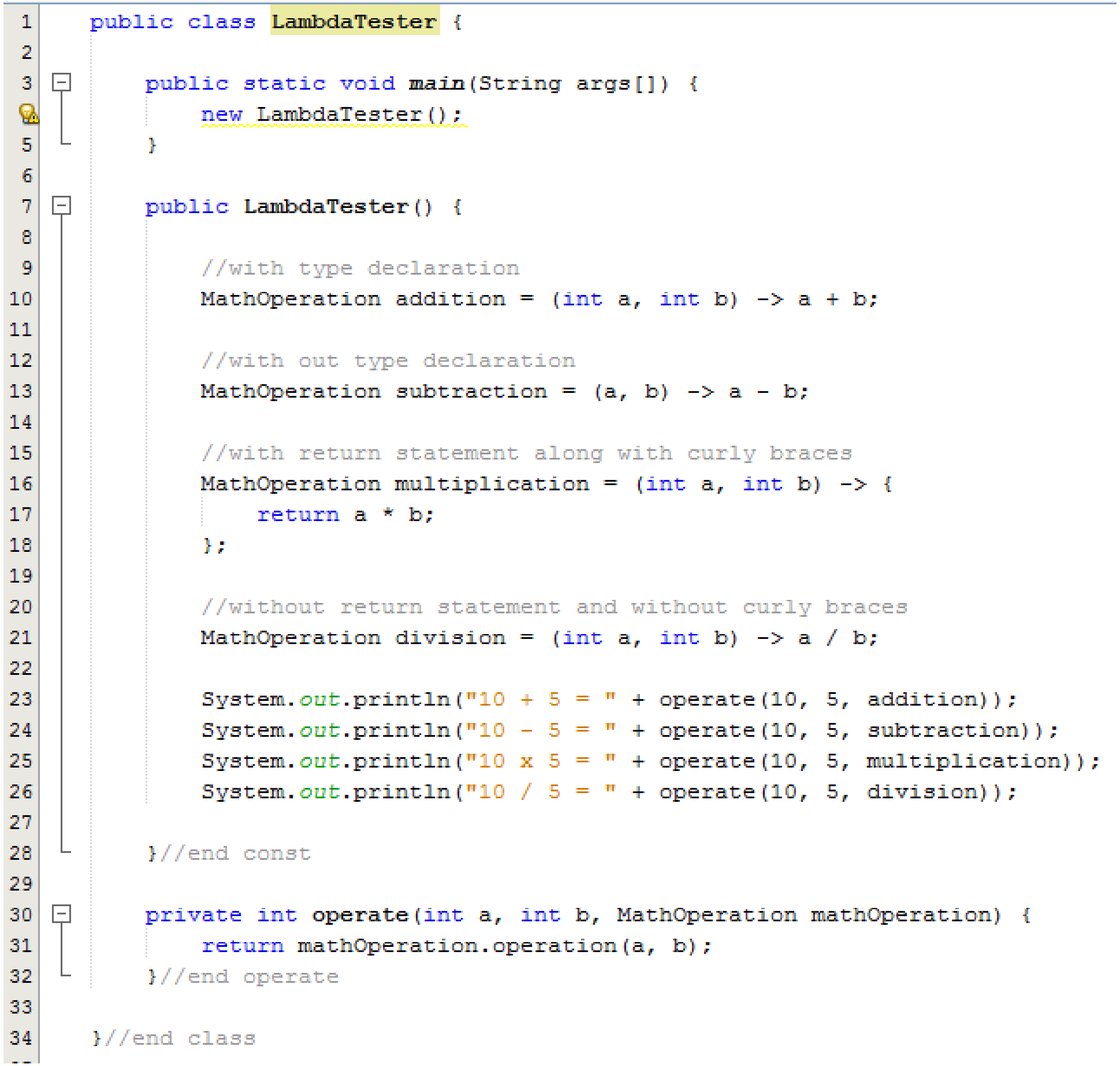
Using Lambdas

- Using anonymous classes you can implement a method that contains the code that’s executed when an event occurs.
- Lambda expressions allow you to do something similar, but with a much cleaner syntax.
    - They allow you to specify code that’s executed without having to create the anonymous class and its method to store this code.
    - 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.

Using Lambdas

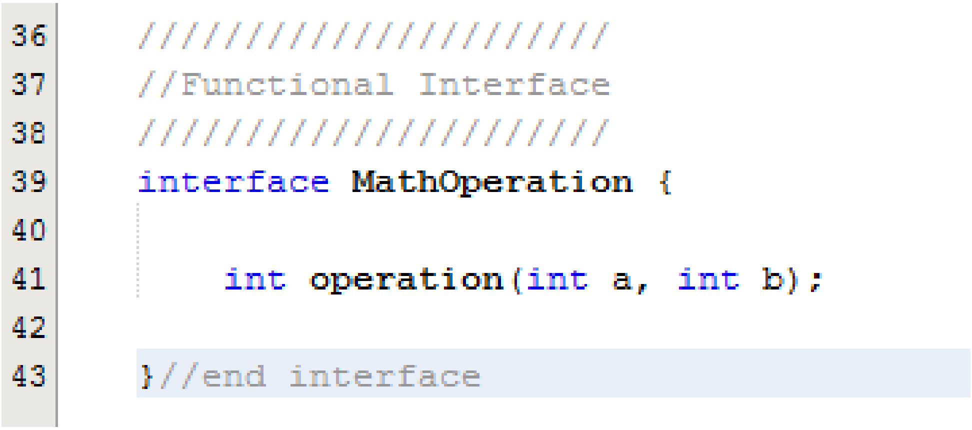
- There are also drawbacks that may mean you do not always use them.
  - Lambda expressions can be difficult to debug because you can’t step through them with the debugger like standard methods.
  - When a lambda throws an exception, the stack trace can be difficult to understand.
  - Methods that use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

Lambda Example

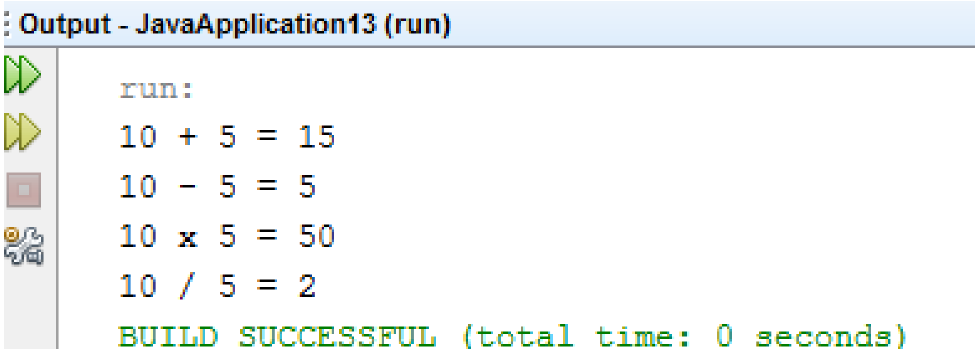


```java
public class LambdaTester {

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

    public LambdaTester() {

        //with type declaration
        MathOperation addition = (int a, int b) -> a + b;

        //with out type declaration
        MathOperation subtraction = (a, b) -> a - b;

        //with return statement along with curly braces
        MathOperation multiplication = (int a, int b) -> {
            return a * b;
        };

        //without return statement and without curly braces
        MathOperation division = (int a, int b) -> a / b;

        System.out.println("10 + 5 = " + operate(10, 5, addition));
        System.out.println("10 - 5 = " + operate(10, 5, subtraction));
        System.out.println("10 x 5 = " + operate(10, 5, multiplication));
        System.out.println("10 / 5 = " + operate(10, 5, division));

    }//end const

    private int operate(int a, int b, MathOperation mathOperation) {
        return mathOperation.operation(a, b);
    }//end operate

}//end class
```

Lambda Example



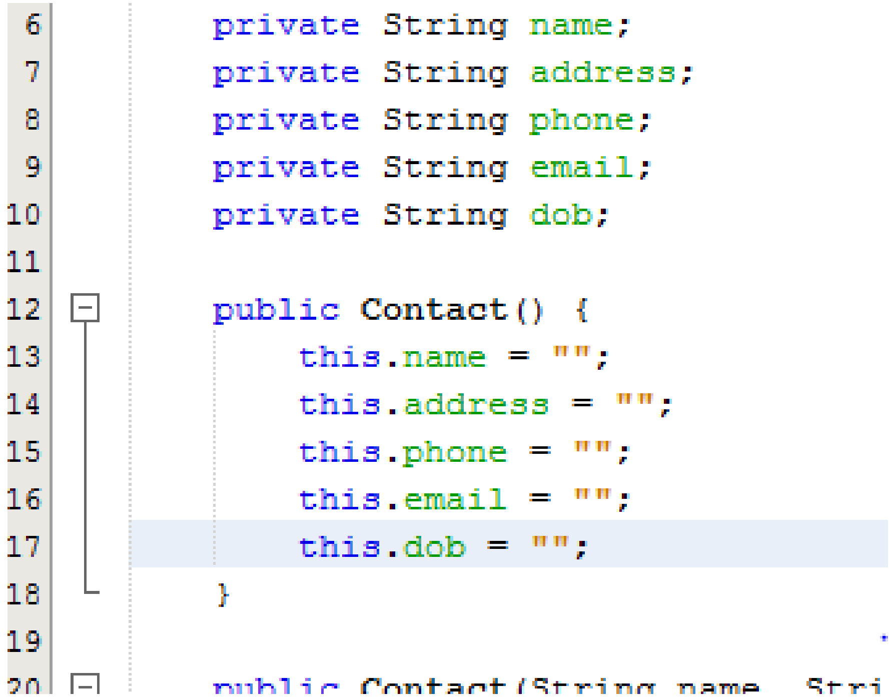
```
//////////////////////
//Functional Interface
//////////////////////
interface MathOperation {

    int operation(int a, int b);

}//end interface
```



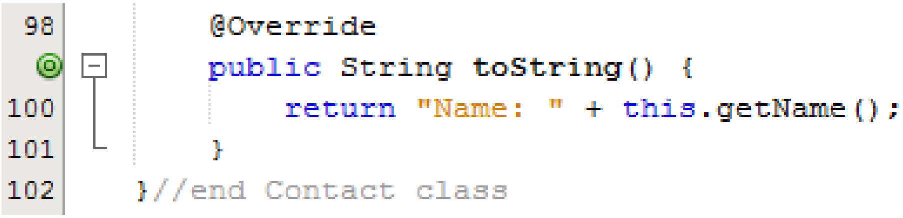
```
run:
10 + 5 = 15
10 - 5 = 5
10 x 5 = 50
10 / 5 = 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Lambda Justification - a method that doesn’t use a lambda expression

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

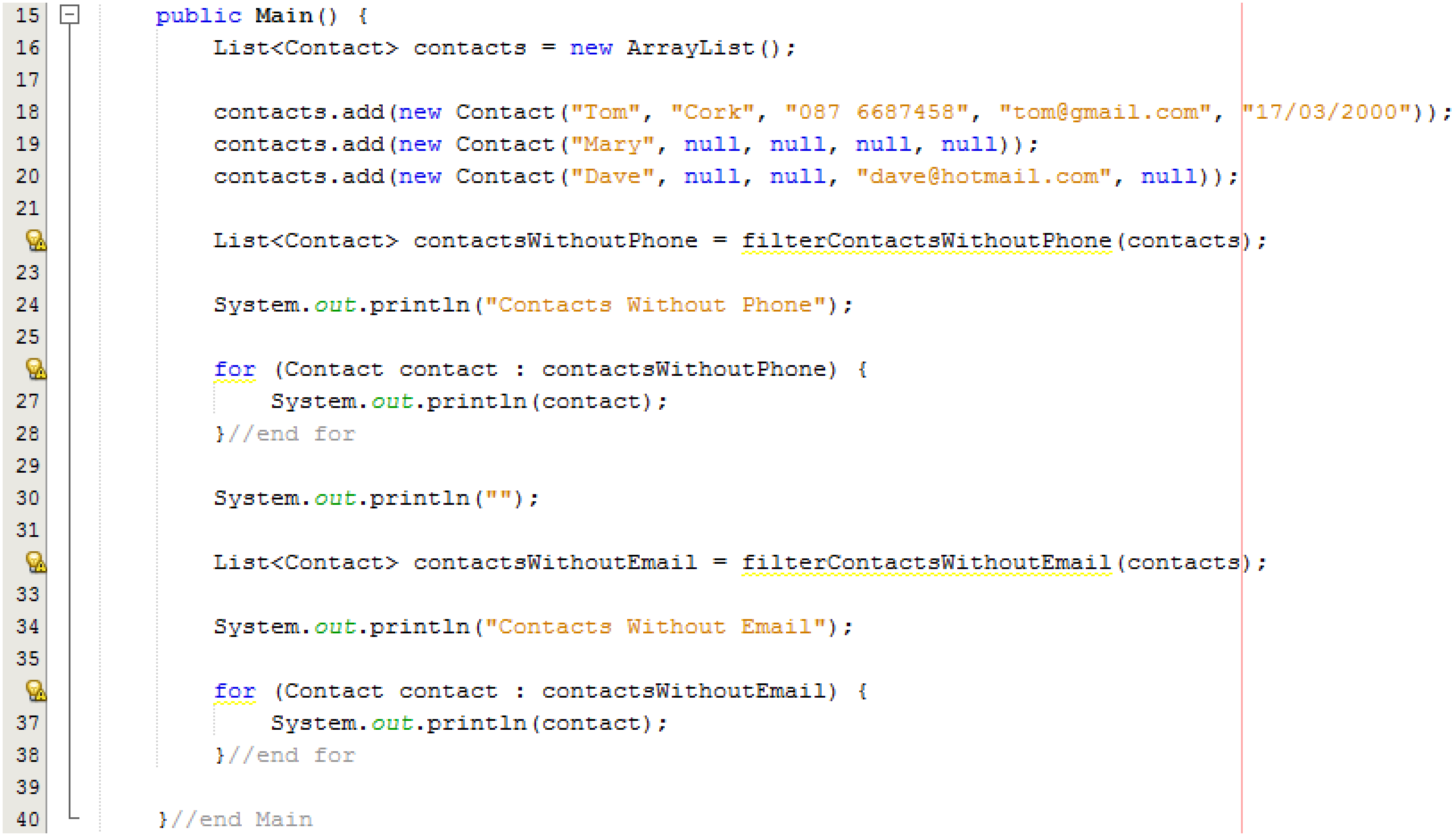


```
    private String name;
    private String address;
    private String phone;
    private String email;
    private String dob;

    public Contact() {
        this.name = "";
        this.address = "";
        this.phone = "";
        this.email = "";
        this.dob = "";
    }

    public Contact(String name, Stri
```



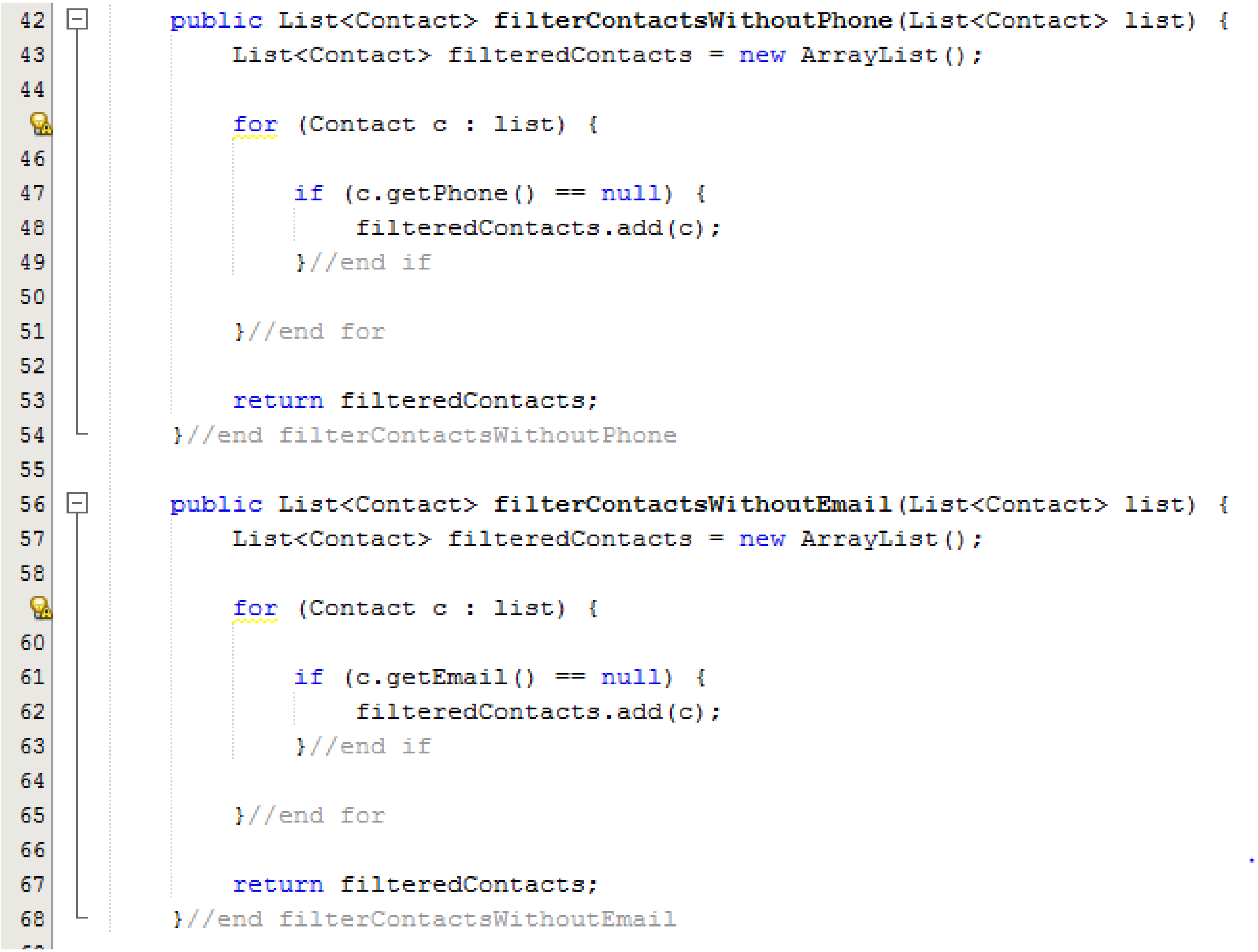
```
    @Override
    public String toString() {
        return "Name: " + this.getName();
    }
}//end Contact class
```

Lambda Justification - a method that doesn’t use a lambda expression



```
    public Main() {
        List<Contact> contacts = new ArrayList();

        contacts.add(new Contact("Tom", "Cork", "087 6687458", "tom@gmail.com", "17/03/2000"));
        contacts.add(new Contact("Mary", null, null, null, null));
        contacts.add(new Contact("Dave", null, null, "dave@hotmail.com", null));

        List<Contact> contactsWithoutPhone = filterContactsWithoutPhone(contacts);

        System.out.println("Contacts Without Phone");

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

        System.out.println("");

        List<Contact> contactsWithoutEmail = filterContactsWithoutEmail(contacts);

        System.out.println("Contacts Without Email");

        for (Contact contact : contactsWithoutEmail) {
            System.out.println(contact);
        }//end for

    }//end Main
```

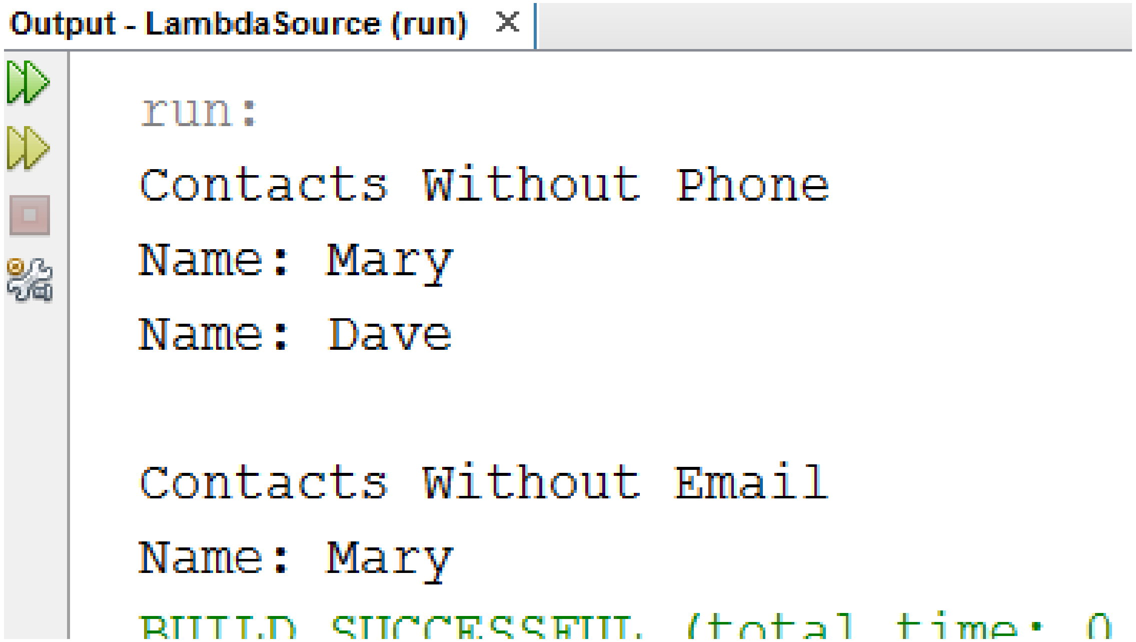
Lambda Justification - a method that doesn’t use a lambda expression



```java
public List<Contact> filterContactsWithoutPhone(List<Contact> list) {
    List<Contact> filteredContacts = new ArrayList();

    for (Contact c : list) {

        if (c.getPhone() == null) {
            filteredContacts.add(c);
        }//end if

    }//end for

    return filteredContacts;
}//end filterContactsWithoutPhone

public List<Contact> filterContactsWithoutEmail(List<Contact> list) {
    List<Contact> filteredContacts = new ArrayList();

    for (Contact c : list) {

        if (c.getEmail() == null) {
            filteredContacts.add(c);
        }//end if

    }//end for

    return filteredContacts;
}//end filterContactsWithoutEmail
```

Lambda Justification - a method that doesn’t use a lambda expression

Output:



```
run:
Contacts Without Phone
Name: Mary
Name: Dave

Contacts Without Email
Name: Mary
BUILD SUCCESSFUL (total time: 0
```

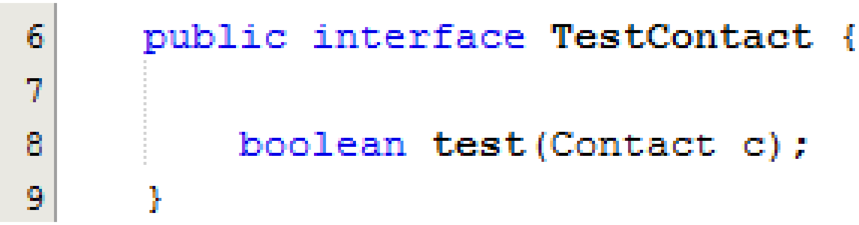
Lambda Justification - a method that doesn’t use a lambda expression

- 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.

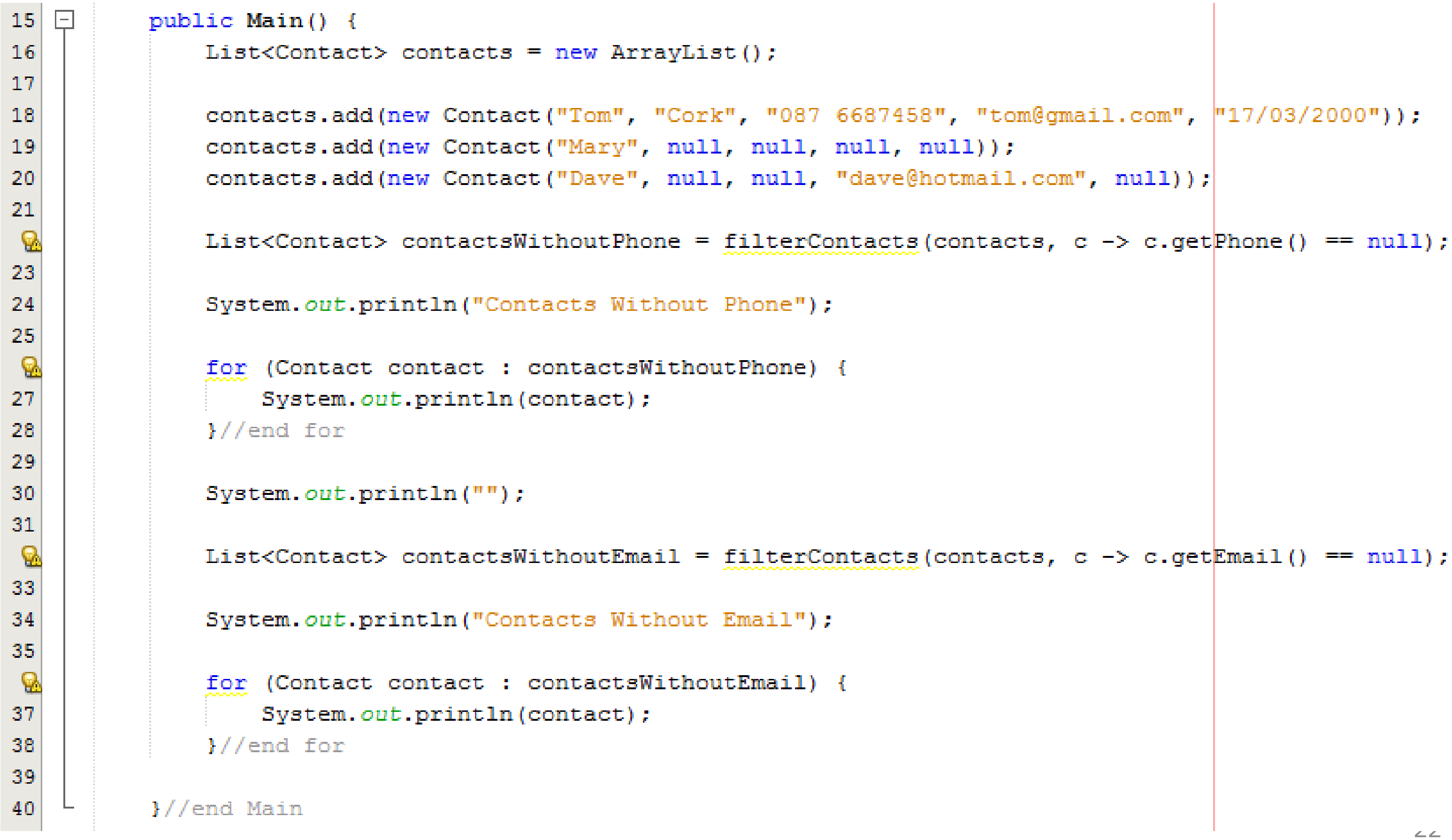
A method that uses a lambda expression

- 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 uses a lambda expression

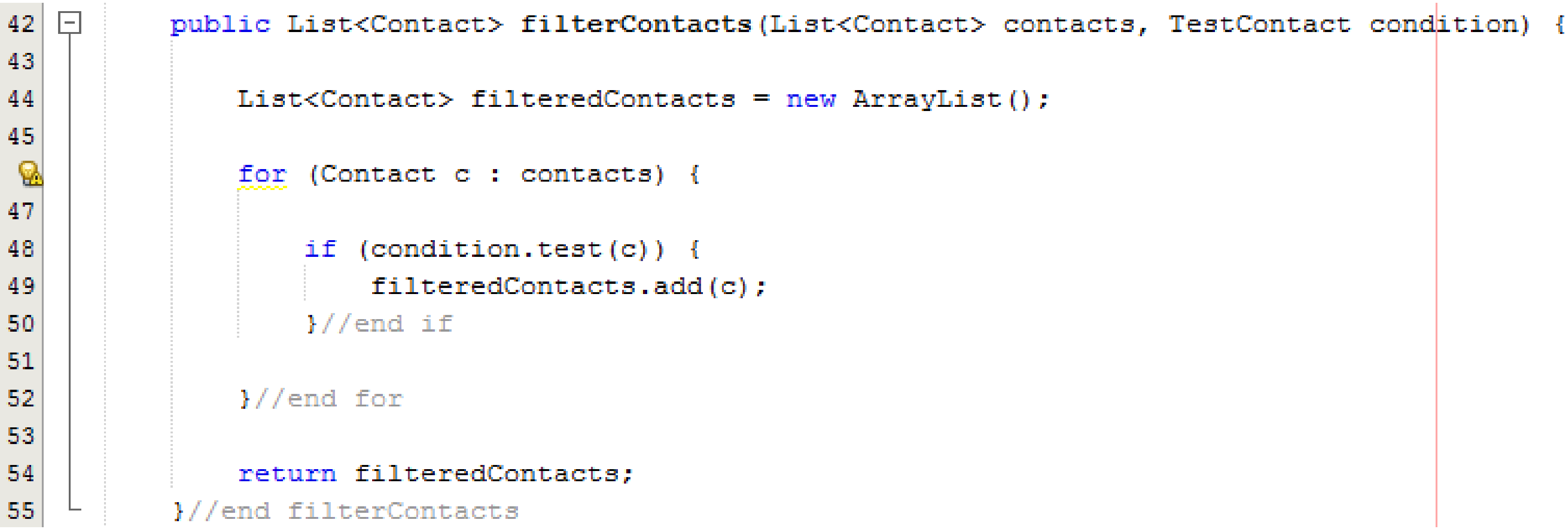


```java
public interface TestContact {

    boolean test(Contact c);
}
```

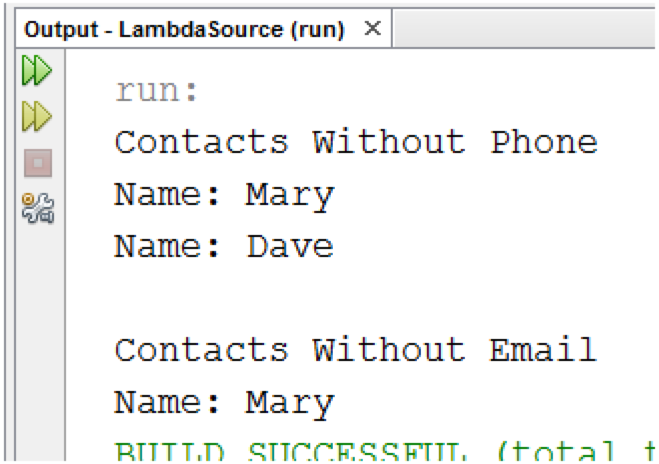


```java
public Main() {
    List<Contact> contacts = new ArrayList();

    contacts.add(new Contact("Tom", "Cork", "087 6687458", "tom@gmail.com", "17/03/2000"));
    contacts.add(new Contact("Mary", null, null, null, null));
    contacts.add(new Contact("Dave", null, null, "dave@hotmail.com", null));

    List<Contact> contactsWithoutPhone = filterContacts(contacts, c -> c.getPhone() == null);

    System.out.println("Contacts Without Phone");

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

    System.out.println("");

    List<Contact> contactsWithoutEmail = filterContacts(contacts, c -> c.getEmail() == null);

    System.out.println("Contacts Without Email");

    for (Contact contact : contactsWithoutEmail) {
        System.out.println(contact);
    }//end for

}//end Main
```

A method that uses a lambda expression



```
public List<Contact> filterContacts(List<Contact> contacts, TestContact condition) {

    List<Contact> filteredContacts = new ArrayList();

    for (Contact c : contacts) {

        if (condition.test(c)) {
            filteredContacts.add(c);
        }//end if

    }//end for

    return filteredContacts;
}//end filterContacts
```



```
run:
Contacts Without Phone
Name: Mary
Name: Dave

Contacts Without Email
Name: Mary
BUILD SUCCESSFUL (total t
```

The output is the same as the first one

A method that uses a lambda expression

- If you want you could code more complex lambda expressions to filter the list in other ways.
- For example, you could check for Contact objects that have a null or empty phone number by using this expression:

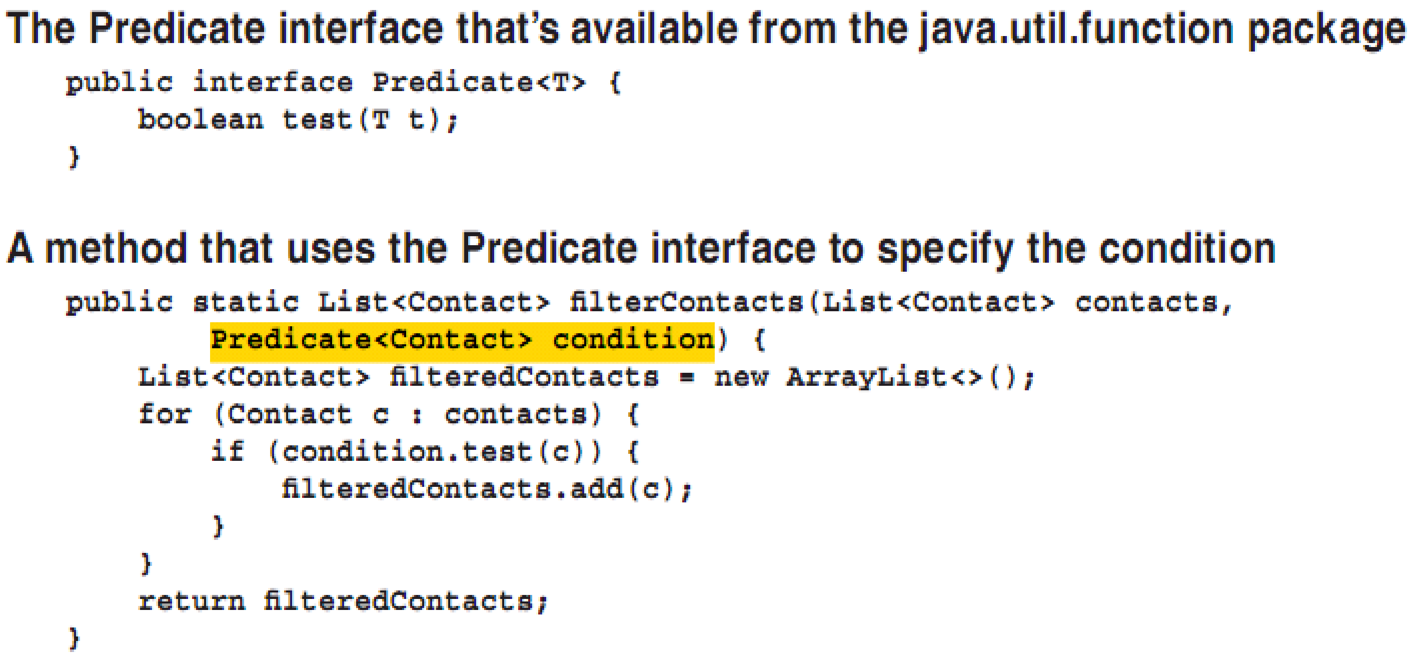
```
c -> c.getPhone() == null || c.getPhone().isEmpty()
```

A method that uses 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.
  - A functional interface that’s available from the java.util.function package.
- This interface defines a method named test that works much like the test method in the TestContact functional interface from the previous example.



The Predicate interface that’s available from the java.util.function package

```
public interface Predicate<T> {
    boolean test(T t);
}
```

A method that uses the Predicate interface to specify the condition

```
public static List<Contact> filterContacts(List<Contact> contacts,
        Predicate<Contact> condition) {
    List<Contact> filteredContacts = new ArrayList<>();
    for (Contact c : contacts) {
        if (condition.test(c)) {
            filteredContacts.add(c);
        }
    }
    return filteredContacts;
}
```

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.

References

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