ALGONQUIN COLLEGE

MODULE 05: ACCESSORY CLASSES AND GENERICS

Professor : Dave Houtman

Office: T323

Office Hrs: Wednesday 14:15 – 15:30

Wednesday (after lecture*)

* confirm beforehand

Email: houtmad@algonquincollege.com

5.1 Copying 1-D Arrays

As described earlier, arrays are declared in the format

```
datatype[] identifier = new datatype[arraysize];
```

where:

datatype is the type of data to be stored in the array, e.g. `int`, `float`, `char`, `boolean`, etc.

identifier is any valid identifier, i.e. it cannot begin with a number, it cannot contain special characters (aside from _ and `$`), etc.

arraysize is the total number of **elements** in the array.

5.1 Copying 1-D Arrays

Note that when you declare an array, using, say

```
datatype[] myArray = new datatype[arraysize];
```

myArray stores a *reference value* only, not the array object itself. A reference value is simply a number used as a handle to the place in memory where the array object is contained.

Therefore, if we have two arrays, called `myArray1` and `myArray2`, of equal type and size, setting

```
myArray2 = myArray1;
```

Does not copy one array into another

does *not* copy the *contents* of `myArray1` into `myArray2`, it only copies the `myArray1` reference value into `myArray2`. Therefore, both array variables contain the same reference value, and point to the same location in memory. *But you have not copied the contents of one array into the other.*

5.1 Copying 1-D Arrays

To copy the contents of one array into another, you could, of course, use a `for` loop, or an *enhanced* `for`, and copy each element from one array into the other.

For example, assume `myArray1` and `myArray2` have the same size and type (assumed to be a `int` in this example). To load the contents of `myArray1` into `myArray2` and print out the contents of the latter, you could use

```
// load myArray2 with myArray1 contents
int ctr =0;
for (int ar1Value: myArray1)
   myArray2[ctr++]= ar1Value;

// print out myArray2 contents
for (int ar2Value: myArray2)
   System.out.println(ar2Value);
```

Liang 7.5 | D&D 7.15

5.1 Copying 1-D Arrays

But there is a better way. Every array created has, in addition to a `length` property, a `clone()` method (overriden from the inherited `Object`, of course). You can simply write:

```
int[] myArray2 = myArray1.clone();
```

to copy the contents of `myArray1` to the new `myArray2`.

That's the way we think about it, but there more to it than that. Technically what happens is: (1) the `clone()` method reserves new space in memory, (2) copies the contents of `myArray1` into that space, and then (3) sets the `myArray2` variable to point to that space. As before, `myArray2` *holds a reference value that points to an array object*, it does not contain the actual array itself.

Since an array is *always* an object in java, it must always contain just a single value, the reference value. Despite appearances, the notation that `myArray1` and `myArray2` are themselves arrays is a convenient fiction: they contain reference values that point to objects in memory which have associated properties and methods that control the memory the array data is stored in.

5.1 Copying 1-D Arrays

There is a second, more powerful way, to copy data from one array to another, using the `System` library. It contains the static method `arraycopy()`, which has the definition:

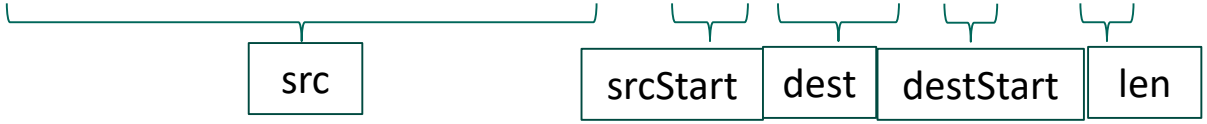
```
System.arraycopy(src, srcStart, dest, destStart, len);
```

where:

`src` is the name of the source array
`dest` is the name of the destination array
`srcStart` is the location of the index to start copying from
`destStart` is the location of the index to copy to
`len` is the number of elements to copy

For example, to copy the numbers 3, 1, 2, 6 from one array to another, use:

```
int[] myAr = new int[4];
System.arraycopy((new int[]{3, 1, 2, 6}), 0, myAr, 0,  4);
```



5.2 Passing Arrays To and From Methods

To pass an array to a method, simply *pass the reference value*, just as you would pass a primitive data. For example:

```
public class loadAndDisplayArray {

    public static void main(String[] args) {
        int[] myIntArray = {3, 1, 2, 6};
        printArray(myIntArray);
    }

    public static void printArray(int[] myArr){
        for (int myValues: myArr)
            System.out.print(myValues + "\t");
    }
}
```

This prints out: **3 1 2 6**

Liang 7.6 | D&D 7.8

5.2 Passing Arrays To and From Methods

Notice the format of the call to the method, and the method header declaration itself. You call this method using:

```
printArray(myIntArray);
```

method name — reference value

The method header declaration looks like:

```
public static void printArray(int[] myAr){
 …
}
```

method name — `myAr` is an array of `int`s

This has the effect of making an assignment between the reference value being passed (`myIntArray`), and the 'declaration' inside the method header itself, i.e.

```
int[] myAr = myIntArray;
```

So if the reference value in `myIntArray` was 47, `myAr` now stores 47.

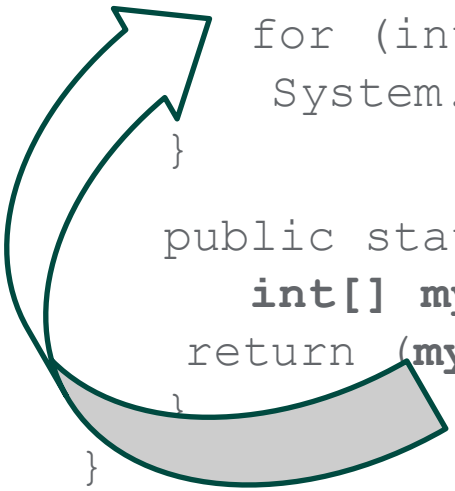
5.2 Passing Arrays To and From Methods

Similarly, to return an array from a method, *pass the reference value* from the instance method back to the calling procedure:

```
public class loadFromArrayAndDisplay {

   public static void main(String[] args) {
      int[] myOutsideRefValue = new int[4];
      myOutsideRefValue = returnNewArray();
      for (int output: myOutsideRefValue)
       System.out.print(output + "\t");
   }

   public static int[] returnNewArray(){
      int[] myInsideRefValue = new int[]{3, 1, 4, 6};
    return (myInsideRefValue);
   }
}
```



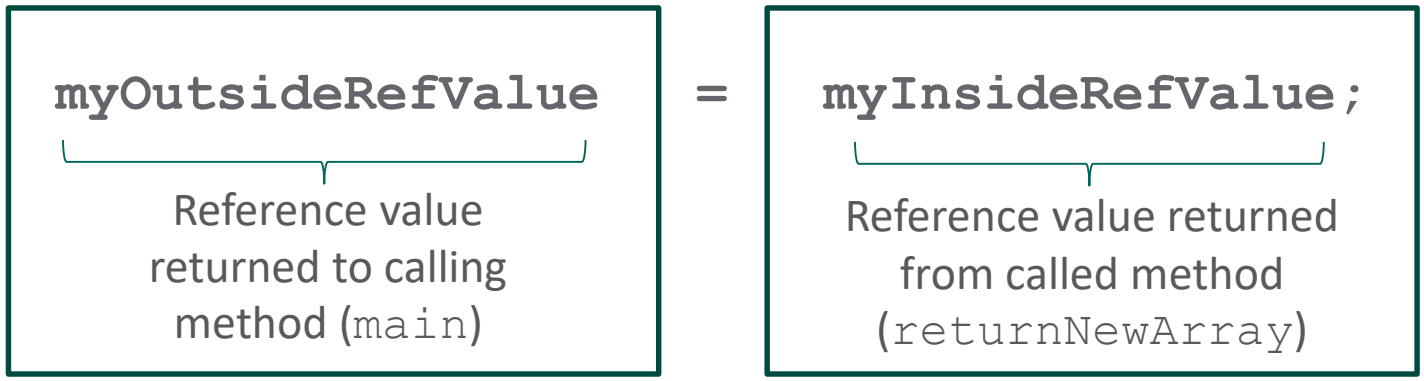
Liang 7.7

5.2 Passing Arrays To and From Methods

Note again the form of the `return` statement. The method is declared as:

```
public static int[] returnNewArray(){…}
```

…this says '`returnNewArray` is an array of `int`s' (reading right to left)—just like a variable declaration. In other words, the method returns a reference value that points to an array of `int`s. So the net effect of the call to `returnNewArray()` is essentially to copy one reference value into another, i.e.



At no point do you actually copy the array, merely the reference value to the array.

5.3 Strings

In Java, a `String` is an object which contains, along with appropriate properties and methods, an array of characters, which remain `private`. As with an array, you can create a `String` with the `new` keyword, like this:

```
String bond007 = new String("Bond, James Bond");
```

But also like an array, `new` is implied when you use the shortcut:

```
String bond007 = "Bond, James Bond";
```

In either case, `bond007` contains a reference value which can be passed to and from a method as a variable of the `String` type.

Liang 4.4 | D&D 14.3

5.3 Strings

The `String` class contains a wide variety of methods, including the ones shown at right.

java.lang.String
+length(): int +charAt(index: int): char +concat(str: String): String +startsWith(prefix: String): boolean +endsWith(suffix: String): boolean +contains(s: CharSequence): boolean +toCharArray(): char[] +toLowerCase(): String +toUpperCase(): String +indexOf(str: String): int +subString(beginIndex: int): String +subString(beginIndex: int, endIndex: int): String +equals(anObject: Object): boolean +compareTo(anotherString: String): int

5.3 Strings

1. With arrays, `length` is a property; with `Strings`, the length of a string is obtained as a method (i.e. its `.length()` not `.length`). See: http://stackoverflow.com/questions/1965500/length-and-length-in-java (retrieved Oct. 14, 2015)

2. You can 'add' one string to another using the `concat()` method. This is a more formal version of the '+' sign, which is an overloaded operator with special meaning in Java when used with strings

3. You can use the shortform operator to append strings. For example:

```
String bond007 = "Bond, ";
bond007 += "James Bond";
```

4. A string is *not* an array of characters, its an object. However, this object can be used to return an array of characters via the `toCharArray()` method.

5. `System.out.println()` is overloaded so that it can handle both `Strings` and character arrays; either works.

5.3 Strings

5. Because any `String` is already an object, you can treat any sequence of characters in double quotes as a string object. So this works:

```
String bond007= "Shaken, ".concat("not stirred");
```

6. Double quotes, " " are used to indicate objects of the `String` type, which is a reference type; single quotes, ' ' , indicate `char`s, a primitive data type. This is an important distinction since:

   a. Strings are passed by reference value, `char`s are passed by value. If you use " " instead of ' ', you'll signal an object type, when what is being passed is a primitive type
   b. Strings may contain extra characters, such as white space; chars do not. So Strings often need to be trimmed first to remove invisible characters that are part of the string by using one of the methods given above, and only then can the individual characters be addressed
   c. A String consisting of a single character is still a string, not a char.

5.4 Immutability

Perhaps the most important feature of a `String` object is that it is ***immutable***; it cannot be changed (or *mutated*) once created: it is `final`. This may be surprising given that you can write something like:

```
String bond007 = "Shaken, ";
bond007 = bond007 + "not stirred";
```

Surely this means that we are appending the second string on to the first, so the reference value points to the same place in memory, but now it points to a longer string?

As before, the use of reference values in Java can lead to some surprising behaviour.

Liang 9.12

5.4 Immutability

It helps to remember that whenever you write a string like "Bond", you are really instantiating a new `String` object, which has its own location in memory. Hence

```
String bond007 = "Shaken, ";
bond007 = bond007 + "not stirred";
```

is the same as:

```
String bond007 = new String("Shaken, ");
bond007 = bond007 + new String("not stirred");
```

resulting in two new locations in memory being reserved. But in fact, there's a third String object created in memory, as we shall see, due to the immutability of Strings.

Liang 10.10

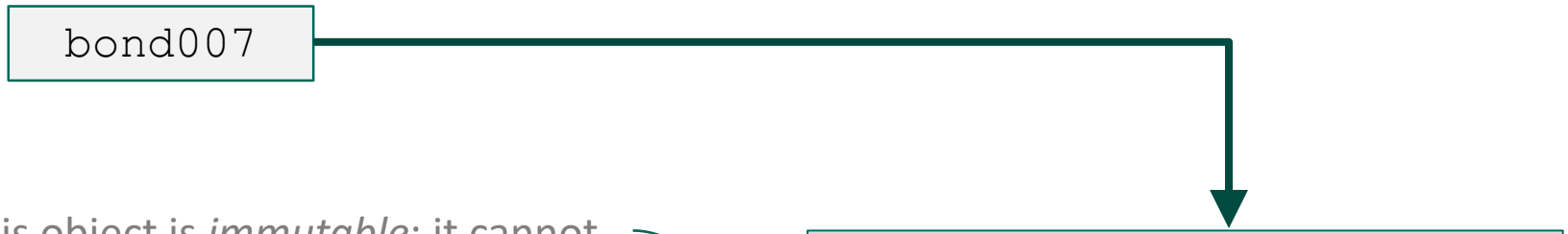
This example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 386, with modifications

5.4 Immutability

Because strings are immutable, they are fixed in memory at the time of their creation, and can only be removed by garbage collection or program shutdown. Thus, if you write:

```
String bond007 = "James Bond";
```

You create a reference value that points to a string with a fixed location in memory:



This object is *immutable*; it cannot be altered while the program is running (except by garbage collection, if it is no longer in use)

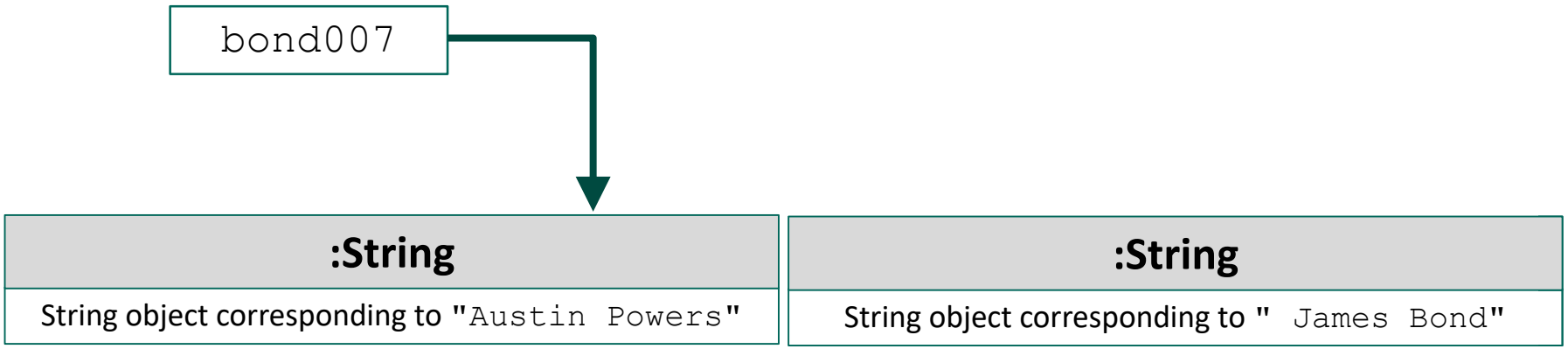
This example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 386, with modification.

5.4 Immutability

If you attempt to put a new string into this location in memory by adding a second command, as in

```
String bond007 = "James Bond";
bond007 = "Austin Powers";
```

you are in fact storing a new reference value in this variable, which points to the new instance of the `String` object containing "Austin Powers“. “James Bond” remains in memory until garbage collection cleans it up.



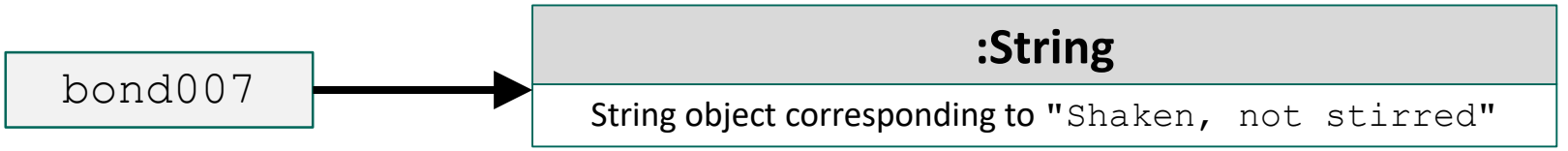
This example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 386, with modifications.

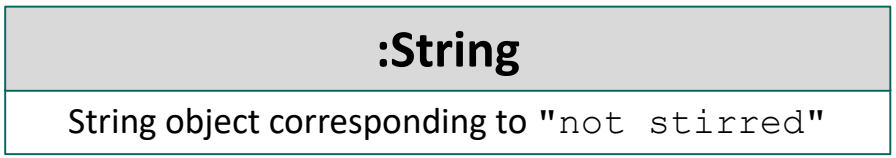
5.4 Immutability

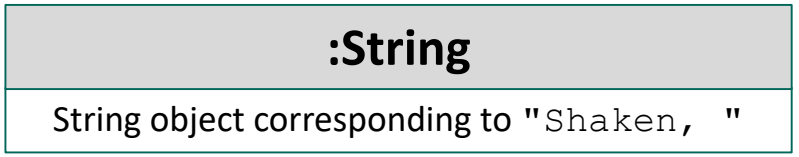
Finally, if you attempt to concatenate two strings, i.e.

```
String bond007 = "Shaken, ";
bond007 = bond007 + "not stirred";
```

the JVM responds by creating a *third* string in memory consisting of the combined strings. And the original two remain unaltered:







This example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 386, with modifictations.

5.4 Immutability

There is a final complication. When you assign two identical strings to different `String` variables, Java attempts to save space by assigning the same reference values to the same place in memory. Thus, when you declare

```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
```

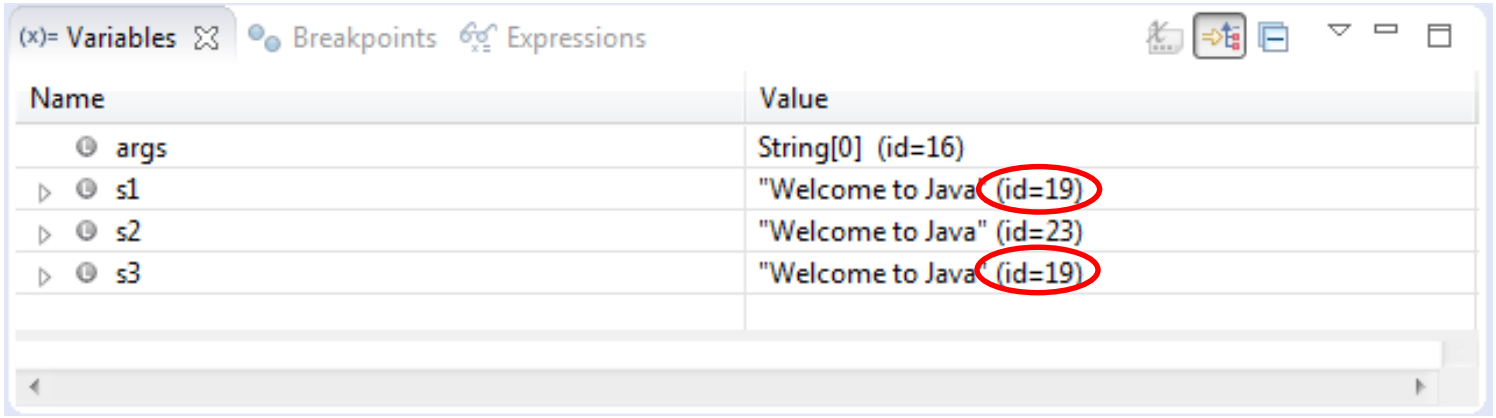
because `s1` and `s3` reference *exactly* the same String, they will be loaded with the same reference value, and hence refer to the same place in memory. But `s2` refers to a new location in memory. Thus:

```
System.out.println("s1==s2 is " + (s1==s2)); // false
System.out.println("s1==s3 is " + (s1==s3)); // true
```

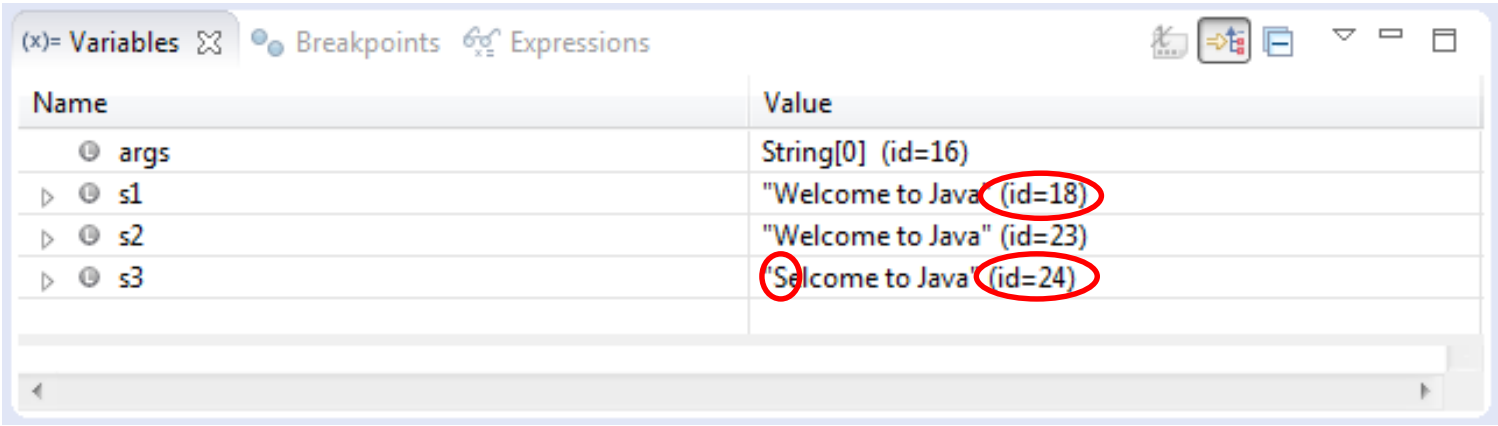
This example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 386, with modifications.

5.4 Immutability

This is made clear in Debug. `s1` and `s3` have the same ID value:



But if we change a single character in `s3`, the ID value changes:



5.5 StringBuilder

Java contains a class, `StringBuilder`, that produces **mutable** string objects; that is, they allow the original object to be manipulated (or mutated—unlike `String` objects). A simplified UML diagram for `StringBuilder` is:

java.lang.StringBuilder
+StringBuilder() +StringBuilder(s: string) +toString(): String +append(data: char): StringBuilder +append(s: String): StringBuilder +delete(startIndex: int, endIndex: int): StringBuilder +insert(index: int, data: char[]): StringBuilder +insert(offset: int, s: String): StringBuilder +length(): int +reverse(): StringBuilder +setCharAt(index:int, ch: char):void

5.5 StringBuilder

For example, the following method reads in a `String`, filters out the non-alphabetic characters or digits (using a static method of the `Character` class), and returns a new `String`:

```
public static String filter(String s){
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < s.length(); i++){
        if (Character.isLetterOrDigit(s.charAt(i))){
            sb.append(s.charAt(i));
        }
    return sb.toString();
}
```

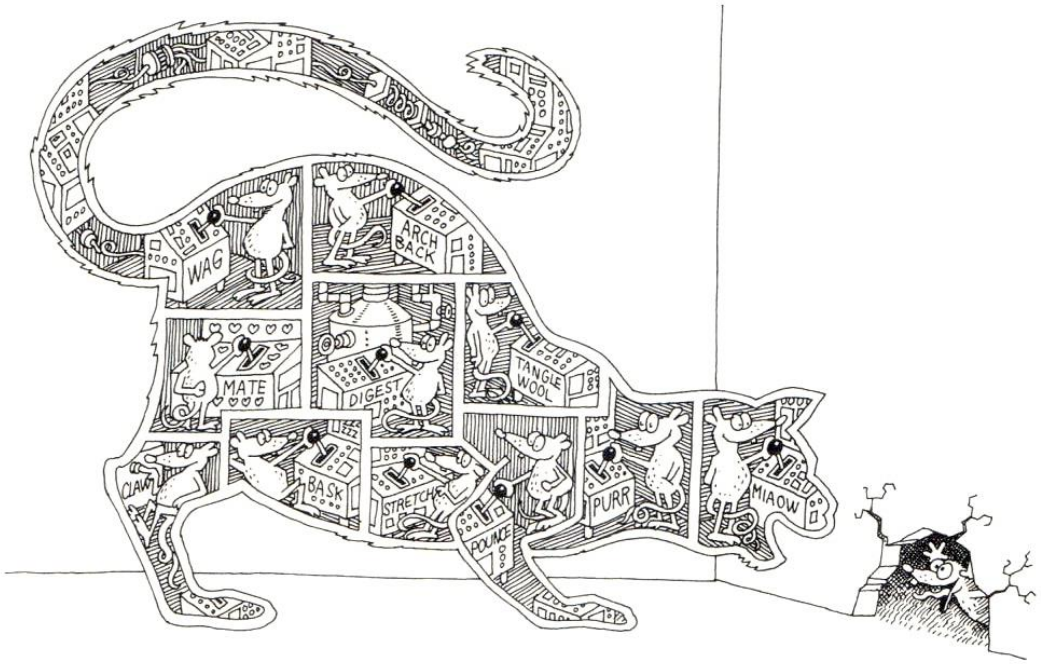
Thus `StringBuilder` is capable of performing most of the operations that `String` is prevented from doing due to its immutability.

Liang 10.11 | D&D 14.4

5.6 Concurrency

Q. ***Why are some objects immutable, and some objects not?***

A. It comes down to the issue of ***concurrency***. The term refers to the fact that several processes in the computer may be operating at the same time.



Concurrency allows different objects to act at the same time.

5.6 Concurrency

A.(con't) If two of these processes are operating on the same piece of data simultaneously (for example, in a database), one process may modify the data in a way not anticipated by the other. In the worst case scenario, a situation called ***deadlock*** can occur, in which neither process is able to 'let go', and the system hangs. This is a potentially serious problem in all multi-threading computer systems. Java's solution is to make some objects unchangeable—*immutable*—so that the problem can't occur. (If you can't change a string, you never need to worry about deadlocking over it.)

The `String` class is one such class.

http://news.nationalgeographic.com/news/2008/10/photogalleries/best-animal-wildlife-photos/photo4.html

5.6 Concurrency

A.(con't)

Since `String` objects are immutable, this can be computationally costly—you can't just add on to a string by tacking on additional characters. But to do otherwise my leave multi-threading versions of your code open to concurrency issues. Java makes certain classes immutable as a way of protecting that data from deadlock at minimum computational cost.

So mutable objects are easier to use, but this comes at the price of possible concurrency issues. Immutable objects are more limited and inflexible in their capabilities, but at least there's no danger of them changing in the middle of an operation, a fact especially important in multi-threading systems.

Having both a `String` class and a `StringBuilder` class allows for both mutable and immutable Strings.*

*See: http://programmers.stackexchange.com/questions/195099/why-is-string-immutable-in-java for an excellent discussion of this issue (downloaded October 9, 2015).

5.7 Arrays of Reference Types

Consider our definition of an array, seen earlier

```
datatype[] identifier = new datatype[arraysize];
```

This works for reference types as well. For example, to create an array of Strings, we'd use

```
String[] seasons = new String[4];
```

This creates an array capable of holding 4 `Strings`. Or to be more precise, this allocates space for 4 reference values of type `String`, where each reference value 'points' to a place in memory. And since this is an array of immutable `String` objects, those actual places in memory holding the Strings cannot be overwritten in code.

Liang 7.2

D&D 7.3

5.7 Arrays of Reference Types

As before, we can initialize our arrays in two different ways, either using the 'full' version:

```
String[] seasons
   = new String[]{"Spring", "Summer", "Fall", "Winter"};
```

Or, we can use the 'shortcut' version of a definition:

```
String[] seasons
   = {"Spring", "Summer", "Fall", "Winter"};
```

In the latter case, as before, `new` is implied.

5.7 Arrays of Reference Types

Since each element of this array returns a reference to a String, we can call on the methods of the `String` object using the index of the array:

```
for(int season = 0; season < seasons.length; season++)
    System.out.print(seasons[season].toUpperCase()+"\t");
```

This prints out:

```
SPRING        SUMMER        FALL        WINTER
```

Alternately, using the *enhanced* `for`, this could be written as:

```
for(String seasonName: seasons)
    System.out.print(seasonName.toUpperCase()+ "\t");
```

with exactly the same result.

5.7 Arrays of Reference Types

This use of reference values applies to more complex classes, i.e. classes containing strings and other classes. Say we have the class:

```
public class BondMovies {
   private String movieName;
   private int yearReleased;
   private double grossEarnings;

   BondMovies(String movieName, int year, double gross){
      setMovieName(movieName);
      setYearReleased(year);
      setGrossEarnings(gross);
   }

   public String getMovieName(){
      return this.movieName;
   }

   public String getMovieYear(){
      return(this.yearReleased);
   } // etc.
}
```

5.7 Arrays of Reference Types

We can then create an array of type `BondMovies`

```
BondMovies[] bonds = new BondMovies[24];
   bonds[0] = new BondMovies("Dr. No.", 1962, 59.5);
   bonds[1] = new BondMovies("From Russia With Love",
                                    1963, 78.9);
   bonds[2] = new BondMovies("Goldfinger", 1964, 124.9);
   …etc.
}
```

Alternately, we could initialize this array using:

```
BondMovies[] bonds ={new BondMovies("Dr. No.", 1962, 59.5),
   new BondMovies("From Russia With Love", 1962, 78.9),
   new BondMovies("Goldfinger", 1962, 124.9),
   … etc.
};
```

5.7 Arrays of Reference Types

We can then reference individual members of the array as follows:

```
System.out.println("Movie Name:" + "\t\t" + "Year:");
for (BondMovies bm: bonds)
    System.out.println(bm.getMovieName() + "\t\t"
                                + bm.getMovieYear());
```

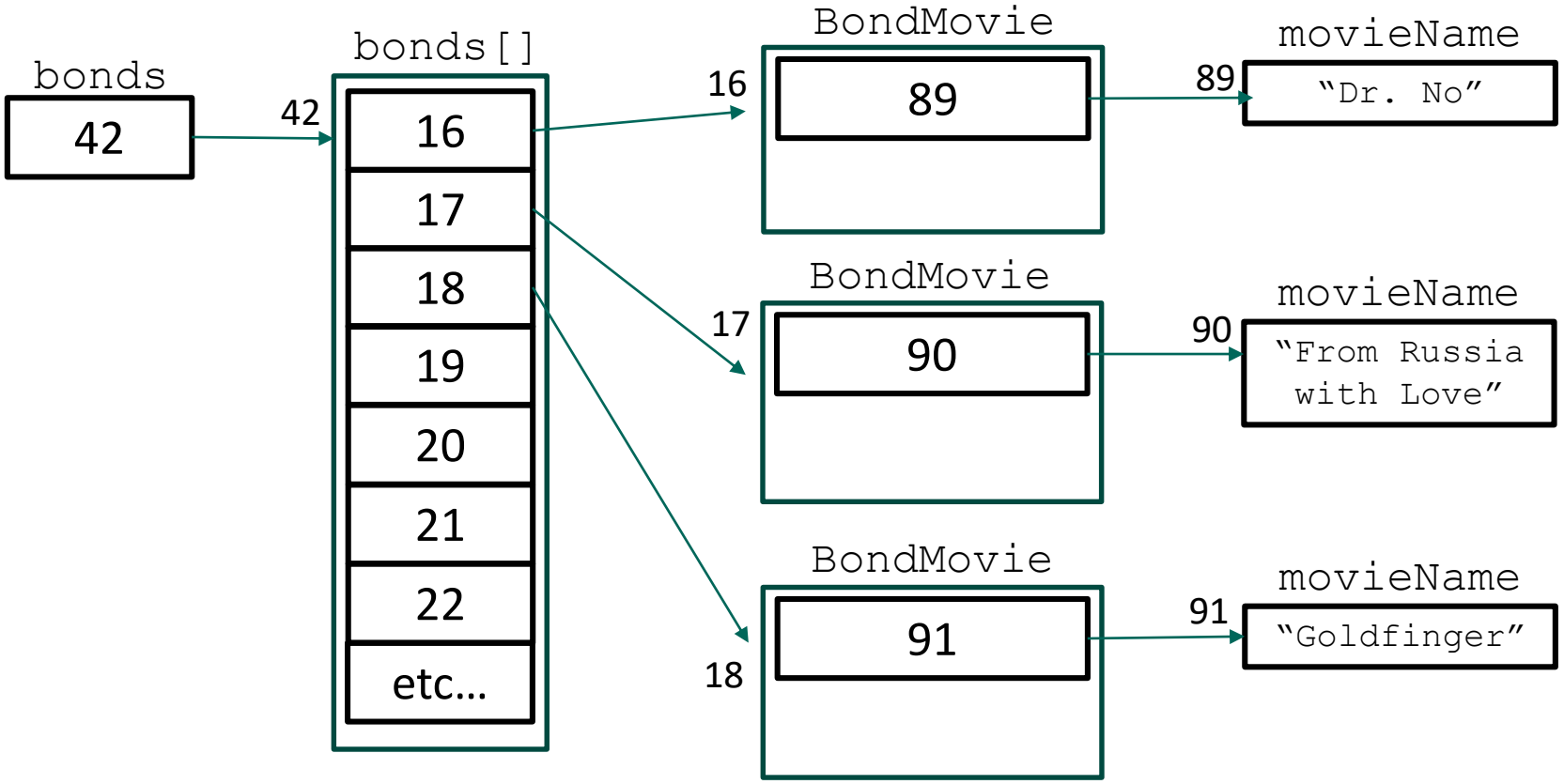
This outputs:

```
Movie Name:                 Year:
Dr. No.                     1962
From Russia With Love       1963
Goldfinger                  1964
Thunderball                 1965
You Only Live Twice         1967
```

…etc.

5.7 Arrays of Reference Types

The important thing to note here is the connection between reference values, and their connection to the objects they hold. (Note: the reference type values are fictitious; they vary each time you run the application). The memory map will look like this:



5.8 The `Arrays` Companion Class

In Java, *arrays are mutable objects;* that is, they can be changed. However, they are not terribly convenient to use in their basic form. So just as `StringBuilder` allows for functionality not found in the `String` class, so also Java's native arrays have a number of companion classes to ease their manipulation.

One such class, called `Arrays`, contains static methods that allow for certain basic operations on arrays, whether they are arrays of primitive or reference data types. The following slide shows a partial list of the methods found in `Arrays`:

Liang 7.12 | D&D 7.15

5.8 The `Arrays` Companion Class

java.util.Arrays
+binarySearch(a: byte[], key: byte): int +binarySearch(a: char[], key: char): int +binarySearch(a: int[], key: int): int *etc.* (for each data type) +sort(a: byte[]): void *etc.* (for each data type) +fill(a: byte[], val: byte): void *etc.* (for each data type) +equals(a1: byte[], a2: byte[]): boolean *etc.* (for each data type) +toString(a: byte[]): String *etc.* (for each data type)

5.8 The `Arrays` Companion Class

So the `Arrays` class allows for basic searching, sorting and comparisons, thus saving you the trouble of having to write and debug the `for` loops needed to perform these operations yourself.

Note that:

1. The `Arrays` class should not be confused with the `Array` class. The former performs simple operations on arrays as just described; the latter uses *Reflection,* a way of determining the names of classes from within a program itself. For now, steer clear of the `Array` class.

2. The `Arrays` class is capable of manipulating arrays of objects, just like simple arrays. However, this requires the use of ***generics***—a way of passing the *type* of class to a method, which uses a new notation in the form `<ClassName>` —the subject of the next section of this module.

5.9 Introduction to Java Generics

The `Arrays` class is capable of copying the elements from one array to another. This includes copying not just primitive data types, but references as well, as shown in the following UML diagram fragment:

java.util.Arrays

```
…
+copyOf(original: boolean[], newLength: int): boolean[]
+copyOf(original: byte[], newLength: int): byte[]
+copyOf(original: char[], newLength: int): char[]
+copyOf(original: double[], newLength: int): double[]
+copyOf(original: float[], newLength: int): float[]
+copyOf(original: int[], newLength: int): int[]
+copyOf(original: long[], newLength: int): long[]
+copyOf(original: short[], newLength: int): short[]
+copyOf(original: T[], newLength: int): <T> T[]
…
```

D&D 20.2

5.9 Introduction to Java Generics

In the highlighted line

```
+copyOf(original: T[], newLength: int): <T> T[]
```

the `copyOf` method of the `Arrays` class clearly takes as arguments an array, `T[]`, and an array size `newLength`. It then copies `newLength` values from `T[]` to another array, also called `T[]` returning a reference value which points to a new array of values of the data type of `T`—exactly what you'd expect for a method designed to copy arrays.

Liang 19 | D&D 20

5.9 Introduction to Java Generics

The *new* part is in the notation:

```
<T> T[]
```

Here, the angular brackets—referred to as the ***diamond notation***— hold the name of the class corresponding to the actual data type (i.e. class) being used. Thus `<T>` stands for a ***generic data type***: *generics let you parameterize types*.

Note:

<T> does not represent the data being passed;
it represents the type of data being passed,
where the type is a reference type, i.e. a class.

5.9 Introduction to Java Generics

While the notation is somewhat different, generic types work exactly the same as when you declare a method, and pass actual values to it.

Specifically, a method declaration has a set of **formal** parameters that appear in the method header. For example, in the following method declaration, **`stringArray`** is the *formal* parameter:

```
public static void printArray(String[] stringArray){
        …
 }
```

while the identifier that you use to pass information to the method is called the **actual** parameter.

```
String[] numbers = {"one", "two", "three"};
printArray(numbers);
```

Here, **`numbers`** is the *actual* parameter.

5.9 Introduction to Java Generics

Generic types work the same way: `<T>` represents a *formal generic type*; it is a dummy value that stands for the actual type to be used. In your code, it is replaced by an *actual generic type*, which will be the class name that you insert into the angular brackets

5.9 Introduction to Java Generics

Thus the UML notation

```
+copyOf(original: T[], newLength: int): <T> T[]
```

says that the `copyOf()` method returns **`newLength`** elements of an array of type **`<T>`** from the initial array **`T[]`**, where **`<T>`** is the formal generic type. So if we wish to copy just the first 5 movies from the `bonds` array, we'd use:

```
BondMovies[] seanConneryBonds =
                                Arrays.copyOf(bonds,      5 );
```

Return type is an array `T[]` of *generic type* `<T>` (i.e. BondMovies)

`original: T[]`

`newLength: int`

Here, `bonds` is the actual generic type corresponding to the formal generic type **`T`**.

5.9 Introduction to Java Generics

The code might be used like this:

```
import java.util.Arrays;
public class Test {
  public static void main (String[] args){
     BondMovies[] bonds = {
        new BondMovies("Dr. No.", 1962, 59.5),
        new BondMovies("From Russia With Love", 1962, 78.9),
        etc.
     };

     BondMovies[] seanConneryBonds = Arrays.copyOf(bonds, 5);
     System.out.println("The first five 'classic' Sean
        Connery Bond Movies are:");
      for (BondMovies scBonds: seanConneryBonds)
         System.out.println(scBonds.getMovieName());
   }
}
```

5.9 Introduction to Java Generics

As with variable names, objects and classes, there is a convention for generic types:

- `E` – Element (used esp. when referring to the elements of an array)
- `K` – Key
- `N` – Number (such as an `Integer` object, `Double` object, etc.)
- `T` – Type (what we will be mostly using to represent classes/interfaces)
- `S`, `U`, `V` etc. – 2nd, 3rd, 4th types, if more than one parameter is generic
- `V` – Value

Remember, these are conventions; actual use will vary.

This information from: https://docs.oracle.com/javase/tutorial/java/generics/types.html (retrieved Oct. 14, 2015)

5.9 Introduction to Java Generics

You might ask why generics are necessary. Prior to JDK 1.5, generics did not exist in Java. If you wanted to pass an object of a certain type to a method, you passed it using the universal `Object` type.

Unfortunately, this meant that *any* kind of object could be passed into a method. *Generics allow you to specify the type of object to be passed at compile time, rather than at run time, when such errors are much harder to detect*. Generics thus improve both your code's reliability and its readability, without adding much real overhead to your code.

See: https://docs.oracle.com/javase/tutorial/java/generics/why.html (retrieved Oct. 14, 2015)

5.9 Introduction to Java Generics

For example, the `comparable` interface allows you to compare two strings. It is used for sorting lists based on criteria you provide. The `compareTo()` method would normally be used for comparing strings and numbers, but prior to Java 1.5, nothing prevented you from doing something like this:

```
Comparable c = new Date();

// several hundred lines of code later
// after you've forgotten what type c is…

System.out.println(c.compareTo("red"));
```

Run-time error

This would trigger an error, because a `Date()` is not an object that can be ordered by `Comparable`. But *the error would only be triggered at run-time, when it would be harder to track down and debug. Caught during the design stage, the bug is much easier to catch.*

Example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 738

5.9 Introduction to Java Generics

As of Java 1.5, you could write this:

```
Comparable<Date> c = new Date();

// several hundred lines of code later
// after you've forgotten what type c is…

System.out.println(c.compareTo("red"));
```

Compile-time error

This generates a compiler error, allowing you to catch the problem much sooner, with fewer debugging hassles.

Example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 738

5.10 The `ArrayList` Class

Generics find wide use in most imported Java classes. `ArrayList` is one such class. Like an array, an `ArrayList` is mutable. But `ArrayList`s have two advantages over standard arrays:

First, `ArrayList`s are flexible; *you don't need to specify the size in advance, as you must when you define an array*. You can arbitrarily add as many elements to an array as you wish, provided they are objects (i.e. have a reference value). And while arrays are 'special' in Java, existing as rather primitive objects the level of the language itself, `ArrayList`s are full-blown objects in their own right.

Liang 11.11 | D&D 7.16

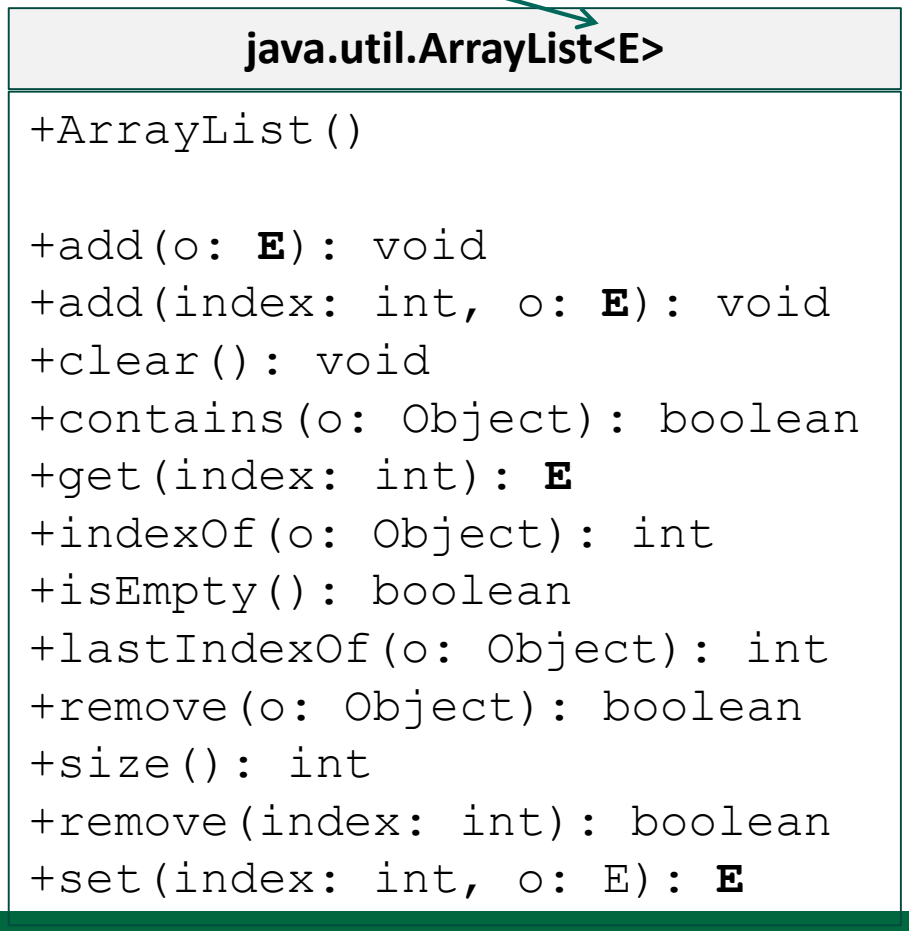
5.10 The `ArrayList` Class

Second, `ArrayLists` come with a full set of methods and fields, so you aren't limited to simple members like `.length` and `clone()` (as you are with arrays). See if you can guess what the following methods do:

java.util.ArrayList<E>
+ArrayList()
+add(o: E): void
+add(index: int, o: E): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): E
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: E): E

5.10 The `ArrayList` Class

Note that the name of the class itself indicates that this class takes a reference to a generic element, `<E>`; this notation in the first compartment warns us that any instantiated `ArrayList` will expect us to specify the type of data used.



5.10 The `ArrayList` Class

So, assuming we've declared `import java.util.ArrayList;`

Ex1: To make an `ArrayList` of `Strings`, we use:

```
ArrayList<String> seasons = new ArrayList<String>();
```

Ex2: To make an `ArrayList` of `Dates`, use could use

```
ArrayList<java.util.Date> dates
                          = new ArrayList<java.util.Date>();
```

Note: you don't need to import a class to use it; you can reference it directly using its import path. Thus you could write:

java.util.ArrayList<java.util.Date> dates = new java.util.ArrayList<java.util.Date>();

Ex3: To make an `ArrayList` of `BondMovies`:

```
ArrayList<BondMovies> bonds = new ArrayList<BondMovies>();
```

5.10 The `ArrayList` Class

Note: as of Java 7, it is no longer necessary to write

```
ArrayList<String> Seasons = new ArrayList<String>();
```

to complete the definition of a new `ArrayList`. Now you can write:

```
ArrayList<String> Seasons = new ArrayList<>();
```

The class type is implicit on the RHS of the assignment; no need to mention `String` twice. This feature is known as ***type inference***.

5.10 The `ArrayList` Class

Here's a comparison of the differences and similarities between an array and an `ArrayList`:

array	ArrayList
`String[] list = new String[10];`	`ArrayList<String> list = new ArrayList<>();`
`list[index]`	`list.get(index)`
`list[index] = "London";`	`list.set(index, "London");`
`list.length`	`list.size();`
	`list.add("Paris");`
	`list.add(index, "Rome");`
	`list.remove(index);`
	`list.remove("London");`
	`list.clear();`

This example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 436.

5.10 The `ArrayList` Class

1. You can only use `ArrayLists` with reference types, not primitive types. Note however that Java contains classes that wrap the primitive data types. So `Boolean`, `Short`, `Float`, `Integer`, etc. are all classes that act like primitive types, but can be used as classes in, for example, `ArrayList`.

So while you *cannot* write:

```
ArrayList<int> list = new ArrayList<>();
```

Can't use primitive data types with Generics

You can wrap an `int` in an `Integer` data type, and write:

```
ArrayList<Integer> list = new ArrayList<>();
int[] myInts = {5, 2, 1, 4, 6, 7, 9};
for (int thisNum: myInts)
   list.add(thisNum);
```

`Integer` is a class, therefore it can be used with generics in place of `int`

5.10 The `ArrayList` Class

2. You can convert from an array to an `ArrayList` by simply using the `Arrays.asList()` method. Given:

```
String[] array = {"red", "green", "blue"};
```

You can convert this to an `ArrayList` using:

```
ArrayList<String> list =
        new ArrayList<>(Arrays.asList(array));
```

This uses a static `Arrays` method to convert from an array to a List that can be used in one of the `ArrayList` constructors.

Note that being able to parameterize the type means that casting is not required, since the compiler knows ahead of time—because you specified the type at compile time—the type of data that was to be used (`<String>`).

5.10 The `ArrayList` Class

2. (con't) Conversely, given the `ArrayList list` above, to create a normal array, use the `ArrayList`'s `toArray()` method.

For example, given an `ArrayList` called `list`, the code

```
String[] arrStr = new String[list.size()];  // new array
list.toArray(arrStr);  //converts ArrayList to array
```

converts the ArrayList `list` into a common array of strings, called `arrStr`

5.11 Generic Methods

Just as generic types can be used to assign type to classes, generic methods can be used *inside* classes, to assign type to methods. An example demonstrates the correct format for such a method.

```
public class GenericPrintTest {

    public static <E> void print(E[] list){
        for (E printThis: list)
            System.out.print(printThis + "    ");
        System.out.println();  // add a line for spacing
    }

    public static void main(String[] args){
        Integer[] integers = {1,2,3,4};
        String[] strings = {"London", "Paris", "Rome"};

        GenericPrintTest.<Integer>print(integers);
        GenericPrintText.<String>print(strings);
    }
}
```

Liang 19 | D&D 20

5.11 Generic Methods

Let's examine this in more detail:

```
public static <E> void print(E[] list){
   for (E printThis: list)
      System.out.print(printThis + "    ");
   System.out.println();
}
```

The method's parameter list indicates that an array of type **E** is to be passed to the method. But the method needs to be 'told' in advance that the parameterized type is called **E**. The correct format for this is to insert the character to be used (remember: the formal type can be any string, but convention dictates we use T, E, etc.) inside the diamond notation just before the return type. So the correct format is:

5.11 Generic Methods

Having declared **E** as the formal type in diamond notation, and indicated that the `print()` method will take as a parameter an array of type **E**, we can then loop through this array, printing out each element (followed by a carriage return at the end of each loop):

```
for (E printThis: list)
   System.out.print(printThis + "    ");
System.out.println();
```

Note the use of `printThis` inside the *enhanced for*. What default method is being used here?

5.11 Generic Methods

Then, for any given type, we call on the method by specifying the type, again inside the diamond notation. This is the type that will be used in calling the method:

```
GenericPrintTest.<Integer>print(integers);
GenericPrintText.<String>print(strings);
```

So when you call the first method using the actual type `Integer`, the `print()` method is compiled as:

```
public static <Integer> void print(Integer[] list){
   for (Integer printThis: list)
      System.out.print(printThis + "    ");
   System.out.println();
}
```

5.11 Generic Methods

Given the arrays,

```
Integer[] integers = {1,2,3,4};
String[] strings = {"London", "Paris", "Rome"};
```

this prints out:

```
1       2       3       4
 London       Paris       Rome
```

5.11 Generic Methods – An Example Using ArrayList

Note that we can avoid overloading using generics. Say we wish to output several different types of arrays of objects, as we did above. Previously, we'd use:

```
1  // Fig. 21.1: OverloadedMethods.java
2  // Printing array elements using overloaded methods.
3  public class OverloadedMethods
4  {
5     public static void main( String[] args )
6     {
7        // create arrays of Integer, Double and Character
8        Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
9        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
10       Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
11
12       System.out.println( "Array integerArray contains:" );
13       printArray( integerArray ); // pass an Integer array
14       System.out.println( "\nArray doubleArray contains:" );
15       printArray( doubleArray ); // pass a Double array
16       System.out.println( "\nArray characterArray contains:" );
17       printArray( characterArray ); // pass a Character array
18    } // end main
19
```

5.11 Generic Methods – An Example Using ArrayList

…where each `printArray()` method is overloaded to handle a specific data type:

```
20      // method printArray to print Integer array
21      public static void printArray( Integer[] inputArray )
22      {
23         // display array elements
24         for ( Integer element : inputArray )
25            System.out.printf( "%s ", element );
26
27         System.out.println();
28      } // end method printArray
29
30      // method printArray to print Double array
31      public static void printArray( Double[] inputArray )
32      {
33         // display array elements
34         for ( Double element : inputArray )
35            System.out.printf( "%s ", element );
36
37         System.out.println();
38      } // end method printArray
39
```

From Dietel & Dietel, pg. 841

5.11 Generic Methods – An Example Using ArrayList

Generics allow us to parameterize the type so that we don't need to rewrite the method for each specific type:

```
21      // generic method printArray
22      public static < T > void printArray( T[] inputArray )
23      {
24         // display array elements
25         for ( T element : inputArray )
26            System.out.printf( "%s ", element );
27
28         System.out.println();
29      } // end method printArray
30   } // end class GenericMethodTest
```

Example from Dietel and Deitel, pg 843

Summary of the Accessory Classes Covered

Class	mutable?	Details
`(array)`	yes	not so much a full class as a core feature of the Java language. It can be used with primitive *and* reference types, but it is limited to very basic functionality that includes `.length` and `.clone()`
`Arrays`	yes	allows for basic sorting and searching of arrays
`ArrayList`	yes	includes a powerful set of methods not found in basic arrays for use with reference types only (via generics), but not primitive types (`int`, `char`, etc.)
`String`	no	allows for basic string manipulation; it does not suffer from concurrency problems, since strings cannot be changed once written. However, its use can be computationally costly exactly because of its immutability
`StringBuilder`	yes	allows for the full range of string manipulation capabilities, however, it is potentially subject to concurrency problems when used with multithreading