Topic 1: The Java Programming Language

Part 2: Packages, Strings, Arrays, and more Objects

Packages

Packages and `import` Statements

- Classes in the Java API are organized into *packages.*
- Explicit and Wildcard `import` statements
  - Explicit imports name a specific class
    - `import java.util.Scanner;`
  - Wildcard imports name a package, followed by an *
    - `import java.util.*;`

- The java.lang package is automatically made available to any Java class.

Some Java Standard Packages

Package	Description
java.io	Provides classes that perform various types of input and output.
java.lang	Provides general classes for the Java language. This package is automatically imported.
java.net	Provides classes for network communications.
java.security	Provides classes that implement security features.
java.sql	Provides classes for accessing databases using structured query language.
java.text	Provides various classes for formatting text.
java.util	Provides various utility classes.

Package

- Java organizes code into packages.
  Ex: `ca.cmpt213.as1` or `com.ibm.db2.qery`
- Set the package:
  `package <reversedDomainName>.<project>;`

  This is the reference to your java file from the **Project root**

- You can then run the files in your package using its fully qualified class name
- Can use code in different package:
  `import ca.sfu.webreg.login;`
  or
  `import ca.sfu.webreg.*;`

(PackageDemo.java)

Example: Create a package

- Create a new Java project in IDE (IntelliJ).
- Create a `Rectangle` class inside a new package.
- Class features
  - Store the **length** and **width** as an `int`s; use constructor to set.
  - Create accessors and mutators for **length** and **width**.

- Create a `Main` class for testing the `Rectangle` class
  - Give it a `main()`.
  - Create `new` function

(Rectangle.java)

(Main.java)

Math

Math Class

- Math class has useful static fields and methods
  - `Math.PI`
  - `Math.pow()`
  - `Math.ceil(), Math.floor(), Math.round()`
  - `Math.abs()`
  - `Math.min(), Math.max(),`
  - `Math.signum(x) // 1.0 if x>0, -1.0 if x<0, 0 if 0.`
  - `Math.random()`
  - `Math.toDegrees(), Math.toRadians()`

String

The `String` Class

- Java has no primitive data type that holds a series of characters.
- The `String` class from the Java standard library is used for this purpose.
- In order to be useful, the a variable must be created to reference a `String` object.

```
String number;
```

- Notice the `S` in `String` is upper case – By convention, class names should always begin with an upper case character.
- Primitive variables actually contain the value that they have been assigned.

```
number = 25;
```

- The value 25 will be stored in the memory location associated with the variable `number`.
- Objects are not stored in variables, however. Objects are referenced by variables.

Primitive vs. Reference Variables

- When a variable references an object, it contains the memory address of the object's location.
- Then it is said that the variable references the object.

```
String cityName = "Surrey";
```

(Ex 1-3)

`String` Objects

- A variable can be assigned a String literal.

```
String value = "Hello";
```

- `Strings` are the only objects that can be created in this way.
- A variable can be created using the *new* keyword.

```
String value = new String("Hello");
```

- This is the method that all other objects must use when they are created.
- Since `String` is a class, objects that are instances of it have methods.
- One of those methods is the `length` method.

```
stringSize = value.length();
```

- This statement runs the `length` method on the object pointed to by the `value` variable.

`String` Methods

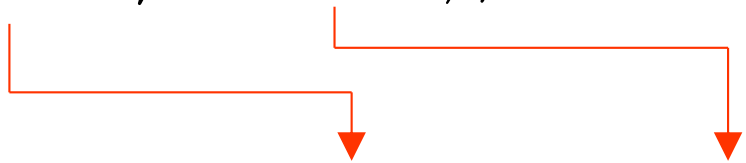
- The `String` class contains many methods that help with the manipulation of `String` objects.
- `String` objects are *immutable*, meaning that they cannot be changed.
- Many of the methods of a `String` object can create new versions of the object. For example:

```
String msg = "H";
msg = msg + "i";
msg += '!';
int count = msg.length();
```

- Some methods on String:
  - `.length()`, `.contains(...)`, `.endsWith(...)`, `.isEmpty()`, `.replace(...)`, `.split(...)`, `.toLowerCase()`, `.trim()`

Returning a Reference to a `String` Object

```
customerName = fullName("John", "Smith");
```



```
public static String fullName(String first, String last)
{
        String name;
        name = first + " " + last;
        return name;
}
```

toString()

The `toString` Method

- The `toString` method of a class can be called *explicitly*:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany.toString());
```

- However, the `toString` method does not have to be called explicitly but is called implicitly whenever you pass an object of the class to `println` or `print`.

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany);
```

The `toString` Method

- The `toString` method is also called implicitly whenever you concatenate an object of the class with a string.

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println("The stock data is:\n" +
  xyzCompany);
```

The `toString` Method

- All objects have a `toString` method that returns the class name and a hash of the memory address of the object.
- We can override the default method with our own to print out more useful information.

(Rectangle.java)

(Main.java)

The `equals` Method

- When the `==` operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.
- All objects have an `equals` method.
- The default operation of the `equals` method is to compare memory addresses of the objects (just like the `==` operator).

The `equals` Method

- The `Stock` class has an `equals` method.
- If we try the following:

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = new Stock("GMX", 55.3);
if (stock1 == stock2) // This is a mistake.
   System.out.println("The objects are the same.");
else
   System.out.println("The objects are not the same.");
```

only the addresses of the objects are compared.

The `equals` Method

- Instead of using the == operator to compare two `Stock` objects, we should use the `equals` method.

```
public boolean equals(Stock object2)
{
   boolean status;

   if(symbol.equals(Object2.symbol && sharePrice == Object2.sharePrice)
      status = true;
   else
      status = false;
   return status;
}
```

- Now, objects can be compared by their contents rather than by their memory addresses.

Methods That Copy Objects

- There are two ways to copy an object.
  - You cannot use the assignment operator to copy reference types
  - Reference only copy
    - This is simply copying the address of an object into another reference variable.
  - Deep copy (correct)
    - This involves creating a new instance of the class and copying the values from one object into the new object.

(Rectangle.java)

Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it

```
public Stock(Stock object 2)
{
   symbol = object2.symbol;
   sharePrice = object2.sharePrice;
}

// Create a Stock object
Stock company1 = new Stock("XYZ", 9.62);
//Create company2, a copy of company1
Stock company2 = new Stock(company1);
```

Pass by Value

Pass by Value

- In Java, all arguments of the primitive data types are passed by value, which means that only a copy of an argument's value is passed into a parameter variable.
- A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.
- If a parameter variable is changed inside a method, it has no affect on the original argument.

(PassByValue.java)

Passing Object References to a Method

- Recall that a class type variable does not hold the actual data item that is associated with it, but holds the memory address of the object. A variable associated with an object is called a **reference variable**.

- When an object such as a String is passed as an argument, it is actually a reference to the object that is passed.

Passing a Reference as an Argument

```
String name = "Bobby";
showLength(name);
```

```
public static void showLength(String str)
{
 System.out.println(str + " is " + str.length() + "
 characters long.");
 str = "Joe";
}
```

`Strings` are Immutable Objects

- `Strings` are immutable objects, which means that they cannot be changed. When the line

  ```
  str = "Joe";
  ```

  is executed, it cannot change an immutable object, so creates a new object.

Passing Objects as Arguments (1 of 2)

- Objects can be passed to methods as arguments.
- Java passes all arguments by value.
- When an object is passed as an argument, the value of the reference variable is passed.
- The value of the reference variable is an address or reference to the object in memory.
- A copy of the object is not passed, just a pointer to the object.
- When a method receives a reference variable as an argument, it is possible for the method to modify the contents of the object referenced by the variable.

Passing Objects as Arguments (2 of 2)

```
displayRectangle(box);
```

```
public static void displayRectangle(Rectangle r)
{
  // Display the length and width.
  System.out.println("Length: " + r.getLength() +
                     " Width: " + r.getWidth());
}
```

(Rectangle.java)
(Main.java)

Returning Objects from Methods (1 of 2)

- Methods are not limited to returning the primitive data types.
- Methods can return references to objects as well.
- Just as with passing arguments, a copy of the object is **not** returned, only its address.

- Method return type:

```
public static BankAccount getAccount()
{
   …
   return new BankAccount(balance);
}
```

Returning Objects from Methods (2 of 2)

```
account = getAccount();
```

```
public static BankAccount getAccount()
{
   …
   return new BankAccount(balance);
}
```

Array and ArrayList

Array

- Arrays have a fixed size when created:
  - `int[] ages = new int[10];`
    `Rectangle[] boxes = new Rectangle[3];`
  - 0 indexed.
- Is Bounds checked!

```
int size = ages.length;
int first = ages[0];
int oops = ages[size];        // Throws exception
```

For-each loop

- The common for loop (still works):

```
for (int i=0; i<arr.length; i++) {
  type var = arr[i];
  statements using var;
}
```

  is now equivalent to:

```
for (type var : array) {
  statements using var;
}
```

ArrayList

- Generic: works with any type of object
- Java includes many generic Collections.
- ArrayList implements the List interface and is backed by an array (fast), and dynamically resizes.

```
List<Rectangle> boxes = new ArrayList<>();
boxes.add(new Rectangle(45.0, 63.0));
for(Rectangle box: boxes) {
    ...
}
```

- Collections only store objects, not primitive types
- To store primitives, use wrapper class
  Integer, Long, Double, etc.