Lecture Notes for Week#2

Recap of Week #1:

Primitive Types:

Recall the primitive types we use for most of our programming:

1. int (32-bits, integer),
2. double (64-bits, floating point number),
3. char (16-bits, a character),
4. boolean (1-bit (*), true or false)

Introduction to Variables:

A variable is the name given to a location in memory. In Java, variable has a type associated with it. A variable is used to store data. In Java, you can only store values with the appropriate type into the variable.

Creating Variables: We call creating a variable "declaring the variable"

A declaration has the form: type variable-name

Two examples: int x

double temperature

Java now sets aside a chunk of memory for each variable. The first is a 32-bit chunk that is associated with the name "x". Java will make sure that only values of type int can be stored in here. (Remember that the type is what WE decide the value represents. The computer does not care. It is happy storing any 32-bit values there.)

The second is a 64-bit chunk of memory that is associated with the name "ratio", and we can store values of type long in it.

Storing Values in Variables: We call storing a value "assigning a variable". To store a value in the variable, you use the = operator.

variable = value

For example: x = 161

temperature = 40.75

This is a source of confusion because the assignment operator looks like math's equality, but it is not an equality test. Instead, we are storing into the memory location named "x" the data that represents the int 161, and we are storing into the memory that is named "temperature" the data that represents the double value 40.75.

Remember that Java is a strong typed language so

x = 3.1415 ← error, you promised x would store int, and you are giving it a double

x = (int)3.1415 ← this is now legal, and it stores value 3 into the location named "x"

To read the value stored in memory at a location, just use the name of the location: x ← this evaluates to whatever value is stored in the memory location in "x", interpreted as data type int

int y

y = x + 50 ← this gets what is stored in x, adds 50 to it, and stores the result in the variable y

Automatic Typecasts (called "type coercions")

Values are automatically converted "narrower" to "wider" types. (Ex: int to double)

Conversion from a "wider" to a "narrower" type requires an explicit typecast. (Ex: double to int)

Generally, when converting to a wider type, Java converts the value to be as close as possible to the original value. When converting to a narrower type, Java generally truncates the value.

Widest: double, float, long, int, short/char, byte: **Narrowest**

Note that short and char are at the "same" level, and so you must explicitly typecast between these two types. Also note that boolean is not in this list. You cannot convert a value of type boolean to any other primitive. You cannot convert another primitive value to boolean.

int y

y = 3.0 ← This is illegal because 3.0 is type double, double is wider than int, and we need an explicity typecast

y = (int)3.0 ← This is legal because the typecast is explicit.

y = 'A' ← This is legal because 'A' is type char, char is narrower than int, so the 'A' will be automatically converted to type int

Interesting Java Behaviors:

Strange behavior 1:

> Integer.MAX_VALUE → 2147483647

> 2147483647 + 1 → -2147483648

Why does Integer.MAX_VALUE + 1 become a large negative number?

- It is because the int type is only 32-bits so there is a limit to the number of values it can store.
- The reason it "wraps" to negative numbers is a trick of the binary representation currently used for int.

Here is another strange behavior, but this time with the double type:

> 1.3 + 1.3 → 1.6

> 1.3 + 2.3 → 3.5999999999999996

Why does 1.3 + 1.3 equal 2.6, but 1.3 + 2.3 does not equal 3.6?

- The numbers are represented in binary, ant some fraction numbers (such as 3/10) can not be represented exactly.
- This is similar to how 1/3 ant 1/7 can not be represented exactly in decimal.
- If the error in the representation is small enough, the Java routine that prints the numbers will round the printed number to a "nice" decimal value.
- For 1.3 + 2.3, the errors in the representation just happen, when added, to be big enough so that the result is not "close enough" to 3.6.

Why should you care?

- Mathematics in Java is not the same as "real" mathematics.
- Errors will crop up due to the fixed size of the numeric data types that would not appear if we were manipulating the numbers using "real" mathematics.
- A programmer must always be alert to situations where such errors could occur so that they do not cause strange program behavior.
- For example, you should avoid testing for equality with floating point values (float or double). If you do any computation on those values, there could be small errors that perturb the value slightly. Instead, it is better to test that the value is within some range. So, instead of testing if a floating-point value is exactly 0, test if it is between -.000001 ant +.000001.

Week #2 Lecture:

Primitive operations and their type rules:

Following are the type rules for the different primitive operators.

1. **arithmetic operators:** +, -, *, /, %
2. **unary arithmetic operators:** +, -
3. **binary operators** (not used in BSM261, they manipulate the binary representations): &, |, ^, ~, <<, >>, >>>

   The above operators have the same type rules:

   a. The two operands (or one operand for the unary operators) must be a numeric primitive type (i.e. not Boolean)
   b. The narrower operand is automatically widened to match the wider operand, and if both operands are narrower than int, they are automatically widened to int.
   c. The result is the type of the wider operand, or int if both operands are narrower than int

Ex: 5.0 + 3 → (the result type is double because of the 5.0, before the addition is performed, the 3 is widened to 3.0)

'a' + 'b' → (the result type is int and both operands are widened to int before the addition is performed)

(short)3 + (byte)5 (the result is type int, and both operands are widened to int before the operation is performed)

4. comparison operators: >, <, >=, <=

The type rules:

a. The operands must be a numeric primitive (a non-boolean primitive)
b. The result type is Boolean
c. The narrower operand type is automatically widened before the operation is performed

Ex: 3 <= 3.1 ← Converts the int 3 to double 3.0 and then compares 3.0 to 3.1

3 < 4 < 5 ← Illegal. First computes 3 < 4, but then the left operand of the second < is type boolean!

5- boolean operators: &&, ||, !, &, | (these are AND, OR, NOT, AND, and OR)

The type rules:

a. The operands (or operand for !) must be Boolean
b. The result is Boolean
c. && and || use "short-circuit evaluation". If the result value is known from just the left operand, the right operand is not evaluated:

AND: &&

left-operand	right-operand	result
true	true	true
true	false	false
false	NOT EVALUATED	false

OR: ||

left-operand	right-operand	result
true	NOT EVALUATED	true
false	true	true
false	false	false

We usually use && and || instead of & and |. The results are the same, and we often take advantage of the short-circuit feature.

For example, suppose we want to test if (y / x) > 1, but this test can cause an error if x stores 0 (divide by 0 error). We can then write the test as:

```
if ((x != 0) && ((y / x) > 1))
```

so that if x stores 0, we get "false" and do not evaluate y / x

Suppose we want to test x < y < z. While that is not valid Java, we can get something equivalent using &&:

(x < y) && (y < z)

6- equality operators: ==, !=

The type rules:

a- The operands can be any type (primitive or non-primitive, and the narrower operand is automatically widened to the wider operand's type.

- it is an error if the automatic type conversion is not allowed

b- The result is Boolean
c- The value is determined by comparing the boolean values of the two operands (after the automatic widening).

Ex: 5 == 5.0 → first converts 5 to 5.0 before comparing

1 != true → illegal because you can't convert between int and Boolean

false != true → legal because both operands are the same type

7- assignment operator: =

a. The left operand must evaluate to a variable. The variable can be any type (primitive or non-primitive).
b. The type of the right operand must be the same or narrower as the type of the variable on the left (*)
c. The result is the same type as the variable and the value that was stored in the variable.

Because = has a value and a type, we can place assignment operators anywhere in our code that expects a value.

Ex: double x

x = 1 ← legal, the int 1 is widened to double 1.0 and stored. The result is type double and value 1.0.

int y

y = 1.0 ← illegal. 1.0 is wider than int

x = y = 3 ← legal. the y = 3 happens first. The result is the int value 3, now we get x = 3, and the 3 is widened to 3.0

The result of this expression will have 3 stored in y, 3.0 stored in x, and the expression will evaluate to the double 3.0

Java Non-Primitive Types (Reference/Compound):

What we have learned so far:

- Everything stored in a computer is just a number in binary.
- All data in a programming language have a type associated with it: the type is what the programmer declares that the data represents.
- Java is strictly typed: all expressions that produce values have types, and Java verifies that each type is used correctly.

There are two kinds of types in Java:

- primitive types (int, double, char, boolean, etc) - these are all pre-defined
- non primitive types (compound types called reference types in your book) - these are both pre-defined and created by the programmer

Non-primitive/Compound types: The basic compound type is called a "class".

A class consists of

1) variables of any type (Java calls variables inside of classes "fields")
2) functions that take 0 or more input, possibly performs some action, and returns 0 or 1 value (Java calls these functions "methods")
3) other reference/compound types (Java calls these functions "nested types")

There are a couple other kinds of reference/compound types that will be covered later.

We will now give examples of 2 pre-defined classes:

Math: a collection of mathematical functions and constants

JFrame: represents a window on your computer screen

JFrame and Math are different kinds of classes. JFrame is a "normal" class in that you can create instances of it.

Math on the other hand, is more limited in that you can't create an instance. (Formally, the Math class has a private constructor. You will see what that means soon.)

Operators for compound/reference types.

Here are all the operators you can do on a compound/reference type. The first and second are the same as for primitive values:

1) The equality operators: ==, !=
2) The assignment operator: =

The others are only for compound/reference types

3) new (used to create an instance/value of a type)
4) . (used to access the fields, methods, or nested types of the class)
5) instanceof (used to test what type a value is)

Terminology: A value of a nonprimitive type is an instance or an object. A class instance is called an "object" in Java.

Important: beginning programmers often struggle with the difference between a type and an instance of the type.

For example, if the type is Movie, instances of the type are "Get Out", "Star Wars: The Last Jedi", "The Devil Wears Prada"

Each instance refers to a specific, separate movie while the type Movie refers to the entire concept of what a film is.

Ex: Suppose we want Movie to be a type.

Movie will contain fields to stored things like: the title, the actors, length in minutes, image data, what part of the film we are currently showing, etc.

Movie will contain methods to do things like: get the title, choose a scene, etc.

Ex:

type	Instance of the type
int	-3, 5, 100
JFrame	All of the windows on your desktop
Movie	"Pilp Fiction", "Lord of the Rings"

What can we do with classes?

1- Create an instance of the class
2- Execute a method of a class
3- Access a field of a class

Using Classes:

1- **To create an instance of a class, use the new operator**

new desired-type(0 or more input values)

new JFrame() → this results in a value of type JFrame

→ note that to use JFrame, you must "import" the package.

The new operator does the following:

- Allocate space in memory for the type instance.
- Initializes the instance using the input data (it does this by calling a special method/function, called a constructor, whose sole purpose is initializing instances)
- Returns the location in memory of the instance. This location is called the "reference" of the instance.

So, when we have

new JFrame()

Java first allocates enough space in memory for the JFrame instance, calls the constructor method with no input, and then returns the location of the instance.

VERY IMPORTANT POINT:

For primitive types, the value of the type is the binary representation of the value.

For reference types, the value of the type is the location of the instance in memory.

Note that every time we call

new JFrame()

we are creating a brand-new instance of a JFrame. If we want to reuse the same instance, we need to remember it. What is the only way to remember anything in Java? With a variable!

Side note: Java Packages

All Java pre-defined compound/reference types are organized into packages. JFrame is in the package javax.swing**.** Math is in the package java.lang**.** You should "import" a reference type before you can use it. import javax.swing.JFrame; new JFrame()

You do not have to import the type. If you don't, you need to use the full name:

new javax.swing.JFrame()

Exception: All types in the package java.lang are automatically imported. As a result, you do not need to import Math to use it.

Now that we can create an instance of a class, let's store the value into a variable.

Variables and Assignment with Reference Types:

The rules are EXACTLY the same as for primitive types

double x ← creates a variable called x that stores values of type double

JFrame frame1 ← creates a variable called window that stores values of type JFrame

x = 5.0 ← stores the value 5.0 in variable x.

frame1 = new JFrame() ← stores the location of the JFrame instance in variable window.

To access something from inside a compound type, we use the "dot" operator.

2- Calling/executing methods of a class

There are two types of methods: instance methods (non-static) and class methods (static).

a. An instance method "acts" on an instance of the class.

To call (execute) an instance method you use:

instance-location.method-name(0 or more inputs separated by commas)

For example, JFrame has an instance method setVisible that takes a single boolean as input. To call the method on the instance whose address is stored in frame1, we use:

frame1.setVisible(true) or frame1.setVisible(false)

Note that the expression to the left of the dot must be an address. It does not have to be a variable. Any expression that gives an address of type JFrame is okay:

new JFrame().setVisible(true)

For another example, JFrame has a method setSize that takes two int values as input. To call the method on the instance whose address is stored in frame1:

frame1.setSize(300, 500)

b. A class method "acts" on the class as a whole and not instances of the class.

To call (execute) a class method you use either:

class-name.method-name(0 or more inputs)

instance-location.method-name(0 or more inputs)

While you can use either, we prefer the first. That make it obvious that you are using a class method. Here are some examples:

Math.cos(1.0) ← the cos method "operates" on the entire Math type.Math.sqrt(5.0)

3- Accessing fields

There are two kinds of fields: instance fields and class fields.

a) **Instance fields:** an instance field as a variable that belongs to an instance of the class.

As a result, each separate instance has its own version of the field. To access an instance field, you use

instance-location.field

Example: JFrame has a field (variable) of type boolean called rootPaneCheckingEnabled

boolean z = frame1.rootPaneCheckingEnabled

frame1.rootPaneCheckingEnabled = true

You can think of the . as an "apostrophe s". The expression frame1.rootPane in Java is asking for frame1's rootPane variable.

b) **Class fields:** a class field is a variable that belongs to the class as a whole.

As a result, there is a single copy of the variable that all instances of the class share.

To access a class field, you use either

class-name.field

instance-location.field

As with class methods, we prefer that you use class-name.field when accessing a class field to make it obvious.

Example, the Math class has a class field called PI that stores the value of PI.

```
double x = Math.PI
```

Class fields are also called "static" fields and class methods are also called "static" methods.

This is not a really good name because it sounds like a class field can't be changed. It can.

A field that cannot be changed is called a "final" field. So Math.PI is a "static final" field. "static" because it belongs to the class and "final" because the value of the field cannot be changed.

Summary:

A compound or reference type contains fields, methods, and other nested types.

The value of a compound/reference type is the address of the data rather than the data itself. (This is why they are called reference types).

The basic compound/reference type is called a class.

The methods, fields, and nested types can either "belong" to an instance of the class (these are called "instance fields" and "instance methods"), or they can "belong" to the class itself (these are called "class fields" or "class methods").

Important Terms to Remember

Here are some common terms you should get used to hearing and using.

1. **non-primitive type:** any type in Java that is not one of the 8 primitives types.
2. **reference type:** another name for a non-primitive type. The name "reference type" reminds us that the value of these types is not the data itself, but it is the location in memory of where the data for the value is stored.
3. **compound type:** another name for a non-primitive type. The name reminds us that a non-primitive type is made up of many other types, variables, and functions.
4. **class:** the most common type of non-primitive type in Java
5. **instance:** a value whose type is a class, or any non-primitive type
6. **object:** another name for an instance. A value whose type is a class, or any non-primitive type.
7. **field:** a variable that is part of a non-primitive type.
8. **method:** a function that is part of a non-primitive type. (In Java, all functions are methods.)

9. **instance field:** a field that belongs to an instance of a type.
10. **class field:** a field that belongs to a non-primitive type (not necessarily a class), and not to the instances of that type.
11. **static field:** another name for a class field.
12. **instance method:** a method that operates on a specific instance of a non-primitive type.
13. **class method:** a method that does not operate on specific instances of a non-primitive type, but on the type itself (not necessarily a class).
14. **static method:** another name for a class method.

Week#3 Lecture Sneak Peek

Java Programming:

1) A simple Java statement ends with a ;
2) A compound Java statement is bracketed by { } and contains 0 or more statements (simple or compound).
3) All Java programming consists of writing compound/reference types. The basic reference type is the class. A class definition consists of a header followed by a compound statement.

```
public class MyFirstClass extends Object {

}
```

- The part starts with "public" is called the "class header".
- The part between the { } is called the "class body". The class body is by definition a compound statement, and all elements of the class (fields, method, other compound types) go in the body (between the { }).

The parts are:

- **public:** the access modifier
- **class:** this is a class
- **MyFirstClass:** the name of the class (a name can be almost anything, but professional Java style is to always start with a capital letter)
- **extends Object:** indicates the super class of this class. Every class has exactly 1 super class (also called the "parent" class).

The access modifier determines where the code can be directly accessed. (This is not a security feature. All code can still be indirectly accessed.)

The access modifier can be any of the four:

- **public:** the code can be used/accessed anywhere in the program
- **protected:** the code can be used in this type and in any type that extends this type (or in any type in the same folder as this type)
- **private:** the code can only be used in the body of this type

If you omit the access modifier, the default is "package": the code can be used in any type that is in the same package (i.e. in the same folder) as the containing type.

What can go in a class body?

- field declarations (these can include assignment operators)
- methods
- other non-primitive type definitions (these are called inner types or nested types)
- constructors: special methods used to initialize instances of the class
- initializers: used to initialize the class itself - these are rarely used in Java programs and so we might not cover them in the course

4) Each public class must go in its own file. The file name must be the same as the class name, and the file extension must be .java. The class must be compiled before you use it. The compiler creates a file with the same name but .class extension that contains the Java bytecode for the class.

5) What's the point of a super class?

- Every class "inherits" all public and protected instance methods of the super classç
- Every class has access to all public and protected constructors, fields, class (i.e. static) methods, and nested types of the super class.

(More about this later.)