ICS 111
Comparisons, Types, Interfaces, Packages

- Review: Object references, comparisons, and equality
- Type operators
- Type parameters
- Java interfaces
- Java packages

Review: Object References

- The toString method of the Object class, called automatically by Java when a String is needed (such as for println) prints out the class of the object and the hash code (memory address) in hexadecimal:

```
public class x {
  public static void main(String[] a) {
    System.out.println(a);
    x b = new x();
    x c = b;
    x d = new x();
    System.out.println(b);
    System.out.println(c);
    System.out.println(d);
  }
}
```

- the result (on my system) is:

```
[Ljava.lang.String;@d716361
x@6ff3c5b5
x@6ff3c5b5
x@3764951d
```

- each object, including the array, has its own address. b and c refer to the same object, so they have the same address
- try this at home!

Review: Equality Comparison

- the == operator evaluates to true if two object references refer to the same object
  - or if both are null, as in

    ```
    if (x == null) { ...
    ```
- the Object equals instance method is the same as ==
- equals methods from other classes may:
  - return true if the objects are ==
  - return true if the objects are != but their contents “match”

```
boolean equals(Object x) {
  if (x == null) { return false; }
  if (this == x) { return true; }
  if (getClass() != x.getClass()) { return false; }
  MyType xx = (MyType)x;
  if (myInstanceVariable == xx.myInstanceVariable) {return true; }
  return myInstanceVariable.equals(xx.myInstanceVariable;
}
```

- the last statement assumes myInstanceVariable is never null – if it could be null, we need an additional test
- if there is more than one instance variable, the last two statements would have to be replaced by code to test equality of all the instance variables, perhaps in a loop

Type/Class Comparisons

- in the previous example we used the getClass() method of Object
- this means we can compare different class objects for equality!
- but remember: with polymorphism, each object may be an instance of more than one class
- so instead of the getClass method, we can use the `instanceof` operator
- to test whether an object is an instance of a specific class:

```
if (obj instanceof Class) { …
```

- this is useful, because otherwise, casting an Object to the type of one of its subclasses may generate a java.lang.ClassCastException
  - to avoid the exception, use instanceof:

```
String a = new String("hello world");
Object b = a;
String c = (b instanceof String ? (String)b : null);
```

- here b is an Object reference referring to a String object, so `b instanceof String` returns true, and `(String)b` casts the object b to a String value
- note that b == c is true because b and c refer to the same underlying object
  - even though be and c have different types!

Type Parameters

- classes such as ArrayList are parametrized on the type of Object that they store
- the class declaration uses a type variable, generally written with a single uppercase letter (T or E are common)
- the generic type T is used in the code as if it were an actual type

```
public class myList<T> {
  private java.util.ArrayList<T> data;
  public myList() {
    data = new java.util.ArrayList<T>();
  }
}
```

- there are some limitations to using type variables – for example, declaring an array of T is complicated

Java Interfaces

- a Java interface is a list of method headers
- a Java class can declare that it `implements` an interface (or more than one)
  - the compiler then checks that the methods in the interface are implemented by the class
- for example, String implements three interfaces: Serializable, CharSequence, Comparable<String>
  - CharSequence requires the charAt() method
- ArrayList<E> implements six interfaces: Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Syntax: Java Interfaces

```
public interface InterfaceName {
  void method1(String arg1);
  String method2();
}
```

- use the keyword `interface` rather than `class`
- all method declarations in an interface are public and abstract
- an interface cannot have static methods
- an interface may declare constants (final variables) with their values
- the keyword `implements` declares that a class implements an interface:

```
public class ClassName implements InterfaceName {
  ...
```

- multiple comma-separated interfaces can be listed

the Comparable Interface

- this is java.lang.Comparable<T>

  ```
  public interface Comparable<T> {
    int compareTo(T);
  }
  ```

- compareTo returns an integer n that is:

  0 if equals() returns true

  n > 0 if this > the argument

  n < 0 if this < the argument

- compareTo can be used to compare objects, rather than just numbers
- the Arrays.sort method can sort an array of any class that implement the Comparable interface
  - this includes String, but does not include Object

using interfaces

- when a method m1 takes a parameter x and calls x.m2()
- it may be a good idea to define an interface `Interface` that only lists the method m2
- the type of the parameter to m1 can be specified as being `Interface`
- example using `Comparable`:

```
public boolean isGreater(Comparable<String> arg) {
  return arg.compareTo("Hello world") > 0;
}
```

- here, the type of the parameter arg is specified using an interface (`Comparable`) rather than a class
  - really, in any type declaration, it’s OK to use an interface wherever a class would be used
- and any object whose class implements `Comparable<String>` can be used as an argument to the `isGreater` method
- for example, it is fine call `isGreater` with a `String` argument

function objects in Java

- Suppose you are implementing a method m1 that operates on a parameter x of type `Object`
- m1 calls a method m2 that depends on the type of x
- if the Object provides m2, all is well: this is what object-oriented programming is all about
  - and is similar to the example on the previous slide, except that the parameter has type Object
- but Object only provides a limited selection of methods. What to do in other cases?
- answer: give m1 an additional parameter y, of a class c (or implementing an interface c) that provides the method m2
- example using a class c that has a method getValue():

```
public boolean isGreater(Object arg, c function) {
  return c.getValue(arg);
}
```

- here, the argument `function` is being used just for the methods it provides access to
- the book has a good example in Special Topic 9.9: when computing an average, we would like to compute an average over arbitrary objects in an array, but to compute an average we need a method (m2) to give us a “measure” or “value” for each object
  - all the objects in an array have the same type, so the same method (c.m2) can give us that measure for every object in the array

Java Packages

- real programs usually include multiple classes in multiple files
- suppose you create a class HelloWorld
- your co-worker creates a different class HelloWorld
- by the end of the development process, you'd like your two programs to work together
- you could always rename one of the two packages, but sometimes that's not so easy:
  - one or both might be in a standard library that you can't change
  - changing either one might require changing lots of other code
- so instead, Java allows you to declare that your class is in a package:

  ```
  package edu.hawaii.esb.example;
  ```

  - the package declaration should come first in a file, and there can only be one per file
  - Java package names should be unique!
- we've seen many packages before, including java.lang and java.util
- source code in the same package is generally found in the same folder (same directory)

Using classes defined in Java Packages

- you have seen this before: `java.util.ArrayList<String>`
  - anything in java.lang (such as `java.lang.String`). is automatically imported
  - in the example on the preceding slide, use `edu.hawaii.esb.example.HelloWorld`
- to make the code more readable, you can import packages:

```
import java.util.*;
ArrayList<String>
```

- you can import any number of packages
  - generally the import statements are all at the top of the source file
    - right after the package declaration (if any)
- too few import declarations make the code very precise, but much longer
- too many import declarations make the code hard to understand for anyone who is not familiar with all the packages

Java Packages: more information

- this material is not in the book. A few references:
  - the wikipedia page Java package
  - the Java tutorials
  - this guide provides many details of package declaration and usage, including access to protected methods and variables by code in the same class
  - and many more!

Summary

- we can do a lot of Java programming without knowing much about memory
- but we do have to understand what it means when two object references refer to the same object
- reviewed .equals and .compareTo
- brief introduction to type comparisons and parametrized types
- interfaces specify what public methods a class provides
  - interface names can be used instead of class names in type declarations
- Java packages allow us to structure our programs in different files and folders/directories
  - and to uniquely identify even classes with the same name