Java Concepts: Classes
Outline

- Class hierarchy and inheritance
- Method overriding and overloading, polymorphism
- Abstract classes
- Casting and instanceof/getClass
- Class Object
- Exception class hierarchy

Reminder of Special Methods and Variables

- a constructor initializes new objects of the given class by setting the values of instance variables
- multiple constructor must have different parameters
- Java automatically calls the no-arguments constructor when the programmer doesn't call a constructor
- interfaces do not list constructors
- accessor methods give access to the values in the instance variables, mutator methods change those values
- `toString` provides a printable representation of the object
- `this` refers to this object
- `super` refers to this object with the type of the superclass

Inheritance

- **A class may `extend` another class, meaning it inherits all of the other class's methods and variables**
  - except for the methods and variables that are `private`!
  - so `protected`, default, and `public` variables and methods are inherited
- **we say that the <u>subclass</u> inherits from the <u>superclass</u>**
  - the child inherits from the parent
- **inheritance means the superclass's variables and methods can be used as if they were declared in the subclass**
- **a class can only extend one other class**
  - if no class is explicitly extended, the new class extends `Object`
  - a class can `implement` any number of interfaces
- **`final` classes such as `String` cannot be extended**

  any other class can be extended

Differences between Interfaces and Inheritance

- a class may <u>implement</u> any number of interfaces, a class can only <u>extend</u> one superclass
- implementing an interface means having to implement all its methods
- inheriting from a superclass means the method implementations are inherited, so <u>don’t</u> need to be re-implemented
- inheriting gives access to protected instance variables
  - interfaces typically only list methods, not variables

Method overriding

- **a class *overrides* a method when it provides another implementation of a method that its superclass already provides**
- **examples:**

    `public String toString()`

    provided by Object, but overridden by many other classes

    `public boolean equals(Object obj)`

    the method provided by Object is the same as ==

    - meaning it returns `true` only if the two expressions refer to the **same object**

    the method provided by String compares the **contents** of the two strings

Method overloading

- **a class *overloads* a method when it provides multiple methods of the same name but with different signatures**
- **examples:**
  - `public String toString()`
  - `public String toString(int numberOfElements)`
  - `public String toString(double maxValue)`
- **real-life examples of overloading:**
  - multiple constructors for the same class
  - in the standard class String:

```
int    indexOf(int ch)
int    indexOf(int ch, int fromIndex)
```

Polymorphism

- *polymorphism* means ``many types”
- polymorphism allows operations to be implemented on the most general possible type (the highest superclass) that supports the operation
- For example, both `Integer` and `Double` are subclasses of `Number`
- in this example, sum can operate on values of type `Number`, because every `Number` has a `doubleValue` method

```
private double sum(Number a, Number b) {
  return a.doubleValue() + b.doubleValue();
}
...
double x = sum(new Double(1.3), new Integer(2));
```

- with autoboxing, the last line can be written as:

```
double x = sum(1.3, 2);
```

- autoboxing automatically creates objects from values of the 8 basic types
  - `int`, `long`, `double`, `boolean`, `short`, `byte`, `char`, `float` are boxed into `Integer`, `Long`, `Double`, `Boolean`, `Short`, `Byte`, `Character`, and `Float`

Polymorphism and Type Conversion

- A variable or expression of a superclass type can always refer to an object of a subclass type:

  ```
  Object obj = new String("hello world");
  ```

  here `Object` is the superclass of `String`

- assigning an object of a superclass type to a variable of a subclass type requires a cast, and may throw a `ClassCastException`:

  ```
  String s1 = (String)obj; // obj refers to a String
  String s2 = (String)(new Object()); //exception!
  ```

Abstract Classes

- **Reminders:**
  - an interface specifies that a class must provide certain methods
  - a class can implement any number of interfaces
  - a class can have only one superclass
- **if the superclass is `abstract`, that means that the superclass itself cannot be instantiated**
  - an abstract class may have abstract methods, which have no implementation
  - a non-abstract subclass <u>must</u> implement all abstract methods of its abstract superclass
  - an abstract class <u>may</u> have non-abstract (i.e. implemented) methods and constructors
- **`Number` is an abstract class – you cannot create a `new Number`**
  - but `Number` does have a constructor, which can be called with `super`

Abstract Class Number

```
public abstract class Number {
  public Number() { ... }// only called as super()
  public byte     byteValue() { return ...; }
  public abstract double   doubleValue()
  public abstract float    floatValue();
  public abstract int      intValue();
  public abstract long     longValue();
  public short    shortValue() { return ...; }
}
```

Class Integer

```
public class Integer extends Number {
  int value = 0;
  public Integer(int value) { super(); this.value =
value; }
  public byte   byteValue()   { return (byte)value; }
  public short  shortValue()  { return (short)value; }
  public int    intValue()    { return value; }
  public long   longValue()   { return (long)value; }
  public double doubleValue() { return (double)value; }
  public float  floatValue()  { return (float)value; }
}
```

Class `Object`

- class `Object` is the superclass of every object class in Java, even if not extended explicitly
- useful methods include:
  - `boolean equals(Object obj);`
  - `String toString();`
  - `int hashCode();`
  - `Class<?> getClass();`
- `hashCode()` will be used later in this course, when hash tables are discussed

What Class does an Object belong to?

- the operator `instanceof` can be used to test class membership, e.g.

```
Integer x = new Integer (99);
if (x instanceof Number) {  // true
  ...
```

- `getClass()` can be used with equals to test whether two objects are in the same class. For example,

```
public class Foo {
  public boolean equals(Object obj) {
    if (this.getClass() != obj.getClass()) {
      return false;
    }
    ...
```

Reminder: Creating New Exceptions

- sometimes we want to create a new exception (shown here with constructors):

```
public class MyException extends RuntimeException {
  public MyException() {
    super();
  }
  public MyException(String s) {
    super(s);
  }
}
```

- A new exception must always extend Exception or RuntimeException
- the call to `super()` is important to initialize the underlying (Runtime)Exception

Exception Class Hierarchy

- `Throwable` is a subclass of `Object`
- `Exception` is a subclass of `Throwable`
- many exceptions are subclasses of `Exception`
- `RuntimeException` is also a subclass of `Exception`
- many common exceptions are subclasses of `RuntimeException`

`RuntimeException`

- exceptions that are subclasses of Exception (but not of RuntimeException) are checked exceptions: if a method may throw a checked exception, the method's declaration must say so:

```
public void
  main(String[] a) throws java.io.IOException {
    ...
```

- only runtime exceptions need not be declared

Example of Different Exceptions

- This example may throw `FileNotFoundException`, which is an `Exception` and must be declared, and also `NoSuchElementException` and `IllegalStateException`, both of which are subclasses of `RuntimeException` and so need not be declared

```
public String readThis() throws java.io.FileNotFoundException
{
  // the constructor may throw FileNotFoundException
  java.io.FileInputStream h = new java.io.FileInputStream("h");
  java.util.Scanner s = new java.util.Scanner(h);
  // nextLine may throw
  // NoSuchElementException or IllegalStateException
  return s.nextLine();
}
```