Java Programming (Basic)

Review

Contents

- Language Basics
- Classes and Objects
- Inheritance
- Polymorphism
- Interfaces
- Packages
- Exceptions
- Numbers and Strings
- I/O
- Conclusion

What is an Object?

- Objects are key to understanding object-oriented technology.
- Real world objects are around us (e.g., dogs, bicycles, cars, houses, tables, people).
- Objects share 2 characteristics:
  - State – the data of interest (e.g., people have a name, hair color, date of birth, etc.)
  - Behavior – what objects do (e.g., people walk, eat, read, etc.)

Software Objects

- Software objects are conceptually similar to real-world objects: They consist of state and related behavior.
  - An object stores its state in *fields* (variables in some programming languages).
  - An object exposes its behavior through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

Software Objects: Benefits

- Modularity
  - The source code for an object can be written and maintained independently of the source code for other objects.
- Information-hiding
  - By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- Code re-use
  - If an object already exists (perhaps written by another software developer), you can use that object in your program.
- Pluggability and debugging ease
  - If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Naming a Variable

- A variable's name is a case sensitive sequence of Unicode letters and digits
- A variable's name must begin with a letter, or the dollar sign '$', or the underscore character '_':
  - _before
  - 2days // error
- By convention, names of variables should begin with a letter (a-z, A-Z), followed by letters, digits, dollar signs, or underscores:
  - miaJima
  - so_desu
  - US$
  - Day_21
  - after_

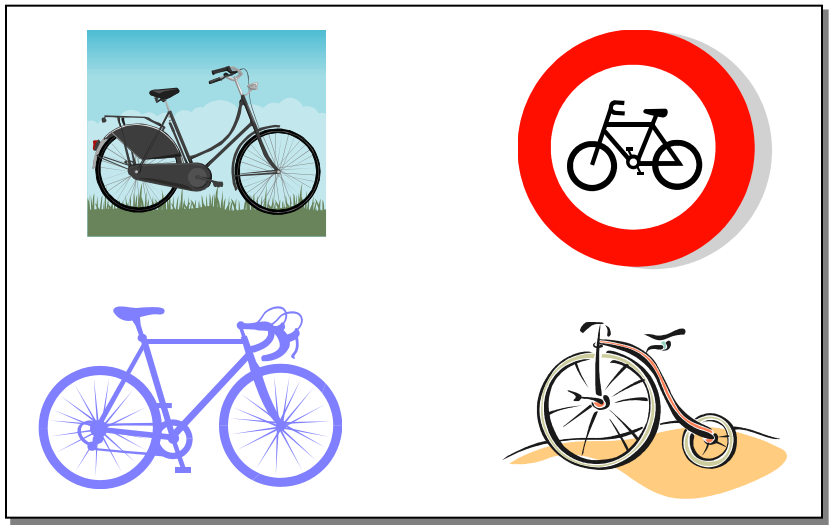
Primitive Data Types

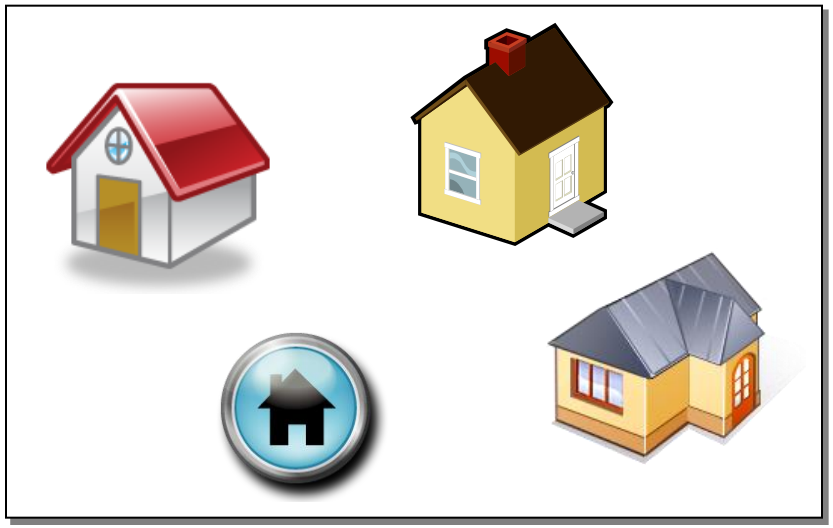
- Java is a *strongly typed* language:
  - All variables must be defined before used
  - The variable's type and name must be stated
- The compiler assigns a default value to an uninitialized field
- The compiler never assigns a default value to an uninitialized local variable
- Using an uninitialized local variable will result in a compile-time error

Primitive Type	Definition	Default Value for Fields
boolean	either *true* or *false*	false
byte	8-bit signed integer	0
char	16-bit Unicode UTF-16 character	'u0000'
short	16-bit signed integer	0
int	32-bit signed integer	0
long	64-bit signed integer	0L
float	32-bit signed floating point	0.0F
double	64-bit signed floating point	0.0D

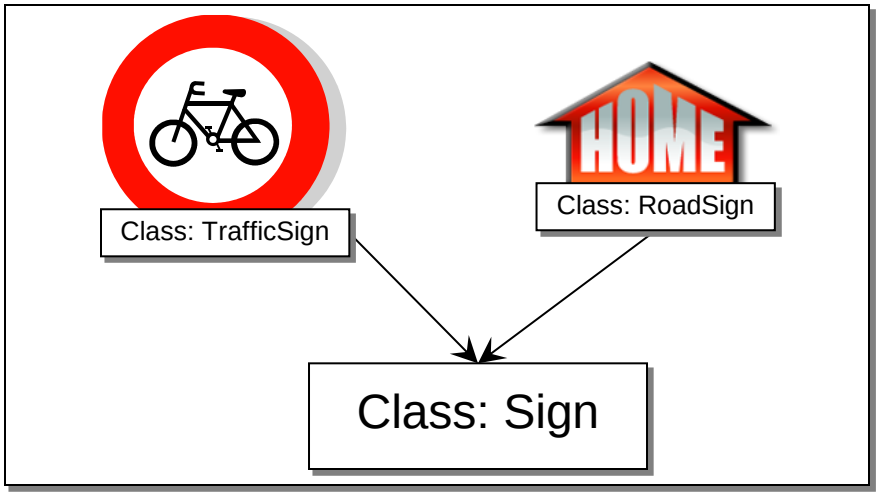
What is Inheritance?

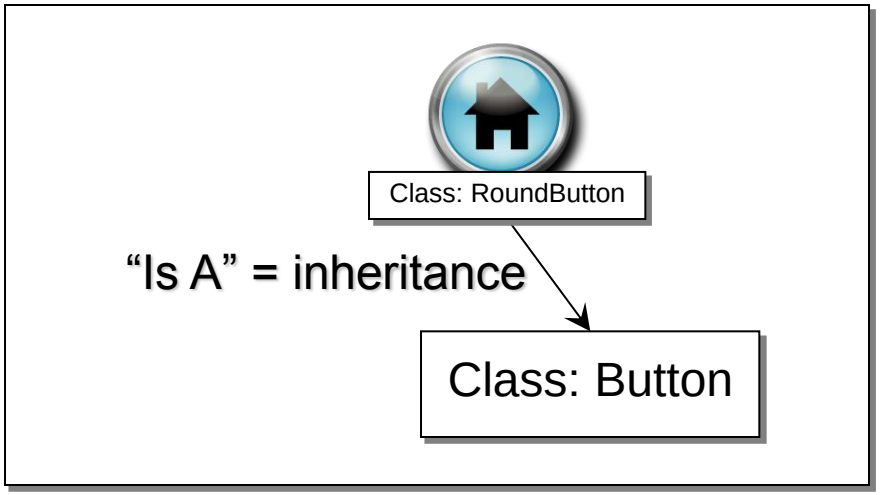
- Similar objects may have something in common
- For example:
  - All bicycles share the same basic features
  - All houses look similar and serve the same purpose
- However, there are different types of bicycles and houses

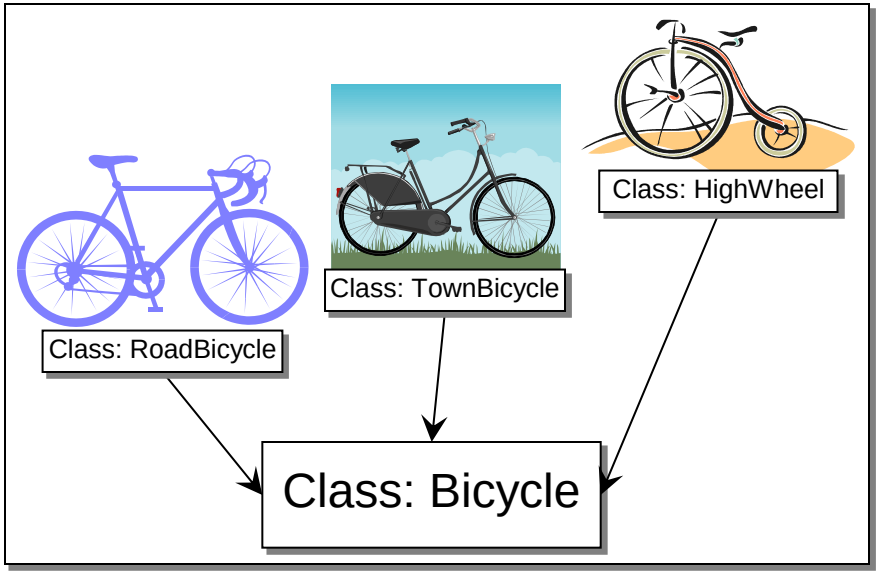


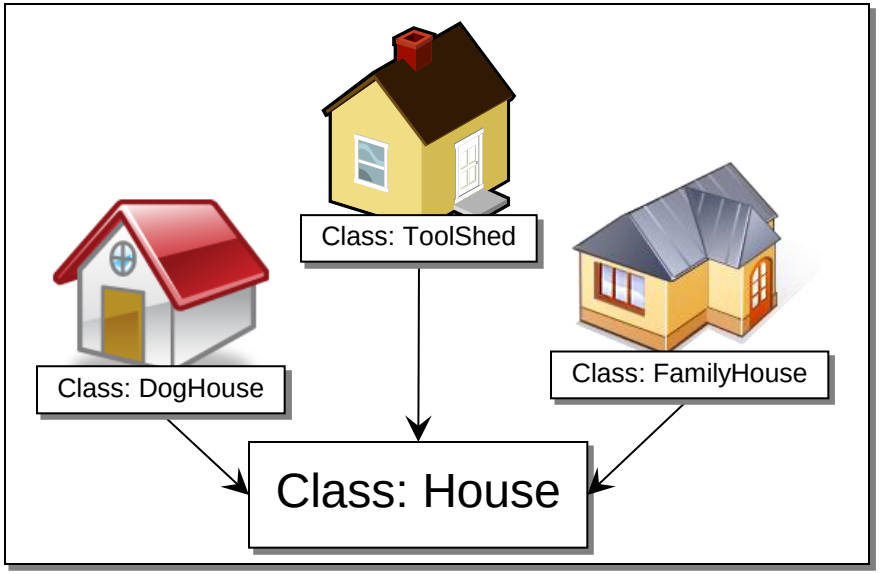


Sharing the State and Behavior









Arrays

- An array is a container that holds a fixed number of values of a single type
- The length of an array is defined upon its creation, and it cannot be changed
- Each item in an array is called an *element*
- Each element is accessed by its numerical *index* (from *0* to *length-1*)

```
int[] ai = new int[5];
Object[] ao = { “1”,”2” };

int aL = ai.length      // = 5
ao.length ≠ 6;          // error

ai[0] = 1;
ai[ai.length-1] = 5;

ao[3] ≠ “6”;            // error
```

Passing Primitive Data Type Arguments

- Primitive arguments, such as an int or a float, are passed into methods *by value*
- If a parameter changes its value in the method, that changed value exists only within the scope of that method
- For the class AA, for example:

```
AA oa = new AA();

oa.m2();
```

```
class AA {

    void m1(int pi) {
        pi += 2;            // → 7
    }

    void m2() {
        int lv = 5;

        m1(lv);  // or m1(5);
        lv *= 2;            // → 10
    }

}
```

Passing Reference Data Type Arguments

- Objects are passed into methods by reference, not by value
- This means that:
  - The reference is passed by value
  - The object's fields may change in the method
  - If a reference parameter changes its value in the method, that changed value exists only within the scope of that method
- For the class AA, for example:

```
AA oa = new AA();
                // oa has the reference

oa.m2();
```

```
class AA {
    int aak;            // = 0, by default

    void m1(AA pa) {
        pa.aak = 5;
        pa = null;
    }

    void m2() {
        AA oa = new AA();
                        // → oa != null
                        // → oa.aak = 0
        m1(oa);
                        // → oa != null
                        // → oa.aak = 5
    }

}
```

Abstract Class

- An abstract class cannot be instantiated:

```
public abstract class AA {
    abstract void method();

}

AA oa ≠ new AA();          // error
```

- An abstract class can be extended:

```
public class BB extends AA {
    void method() {
       …
    }

}

BB ob = new BB();
```

- A class may be declared abstract to prevent its instantiation:

```
public abstract class CC {
    void method() {
       …
    }

}

CC oc ≠ new CC();          // error
```

Abstract Method

- An abstract method is declared without an implementation
- If a class has abstract methods it must be declared abstract
- A class that extends an abstract class and implements the abstract methods is not abstract
- If a class inherits an abstract method but does not implement it, it must be declared abstract

```
public abstract class AA {
    abstract void method();

}

public class BB extends AA {
    void method() { … }

}

abstract class CC extends AA {
    void method(int i) {
            …
    }

}
```

Example: *final* Method and Class

```
public class AA {
    private int aak;

    final void method() {
            …
    }

}

class BB extends AA {

    void method() { … }

}
```

```
public final class AA {
    private int aak;

    void method() {
            …
    }

}

class BB extends AA {

}
```

Example: Polymorphism

```
class Person {
  private String name;
 public Person(String name) {
   this.name = name;
 }
 public String introduction() {
    return "My name is " + name + ".";
  }
}
class Student extends Person {
  private String id;
  public Student(String name, String id){
    super(name);
    this.id = id;
  }
 public String introduction() {
    return "I am a student. " +
    super.introduction() + " My ID is "+ id + ".";
  }
}
```

```
public class PolymorphismDemo2 {
  public static void main(String[] args) {
    m(new Student("Saito", "s115333"));
    m(new Person("Tanaka"));
  }
  public static void m(Person x) {
    System.out.println(x.introduction());
  }
}
```

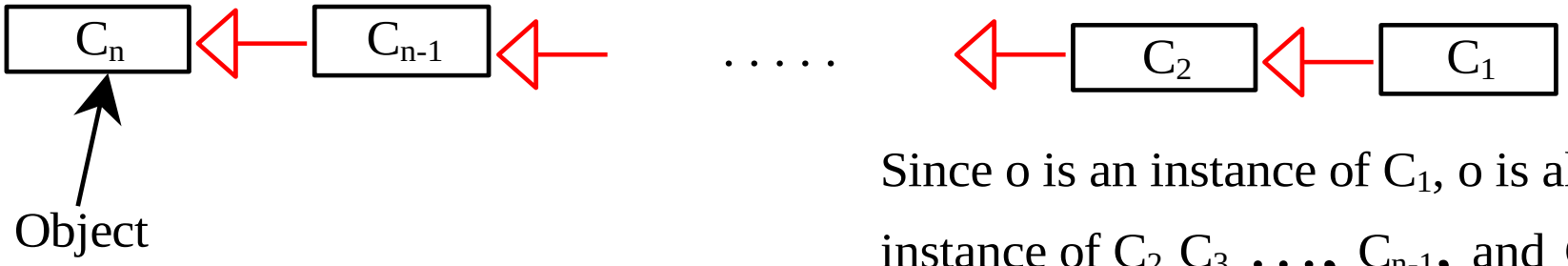
- Output of this program:
  - I am a student. My name is Saito. My ID is s115333 .
  - My name is Tanaka.

Comments on the Previous Slide

- In programming, *polymorphism* is the ability for same code to be used with several different types of objects and behave differently depending on the actual type of object used.
- Method *m* takes a parameter of the *Person* type. An object of a subtype can be used wherever its supertype value is required.
  - This feature is known as *polymorphism*.
- When the method *m(Person x)* is executed, the argument x's *introduction* method is invoked. *x* may be an instance of *Student* or *Person*. Classes *Student* and *Person* have their own implementation of the *introduction* method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.
  - This capability is known as *dynamic binding*.

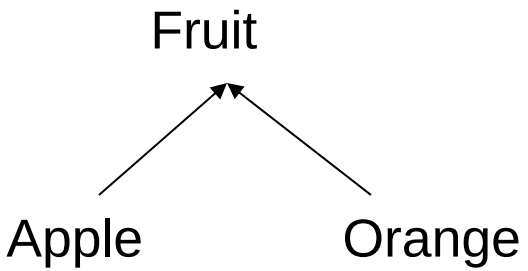
Dynamic Binding in Java

- Java methods are polymorphic by default they are bound at run time.
  - *static* or *final* (*private* methods are implicitly *final*) are bound at compile time (*static binding*).
- We can conceptually think of the dynamic binding mechanism as follows: Suppose an object *o* is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n.
- That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the *Object* class.
- If *o* invokes a method *p*, the JVM searches the implementation for the method *p* in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



TIP

- To help understand casting, you may consider the analogy of fruit, apple, and orange with the *Fruit* class as the superclass for *Apple* and *Orange*.
- An apple is a fruit, so you can always safely assign an instance of *Apple* to a variable for *Fruit*.
- However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of *Fruit* to a variable of *Apple*.




```
Fruit f;
Apple a = new Apple();
Orange o = new Orange();
f = a; // implicit casting, up-casting
f = o; // implicit casting, up-casting
If (f instanceof Apple) {
   a = (Apple)f;    // explicit casting
                    // down-casting
}
```

What is an Interface?

- In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only*
  - constants,
  - method signatures, and
  - nested types.
- There are no method bodies.
- Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.
- A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface BicycleInterface {
        void changeCadence(int newValue);
        void changeGear(int newValue);
        void speedUp(int increment);
        void applyBrakes(int decrement);
}
```

What is an Interface?

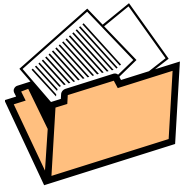
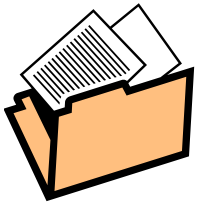
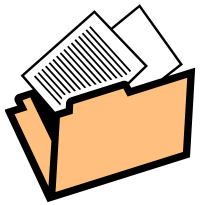
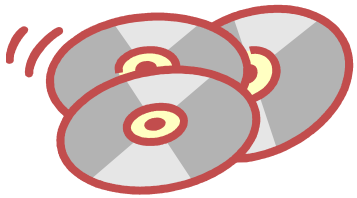
```
class ACMEBicycle implements BicycleInterface {
      private int cadence, speed, gear; // three fields  represent the object state
      public ACMEBicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;             // the ACMEBicycle class has one constructor
        cadence = startCadence;
        speed = startSpeed;
      }
      void changeCadence(int newValue) {// methods define interactions of
           cadence = newValue;                // the object with the outside world
      }
      void changeGear(int newValue) {
           gear = newValue;
      }
      void speedUp(int increment) {
           speed = speed + increment;
      }
      void applyBrakes(int decrement) {
           speed = speed - decrement;
      }
      void printStates() {
           System.out.println("cadence:"+cadence+" speed:"+speed+" gear:"+gear);
      }
}
```

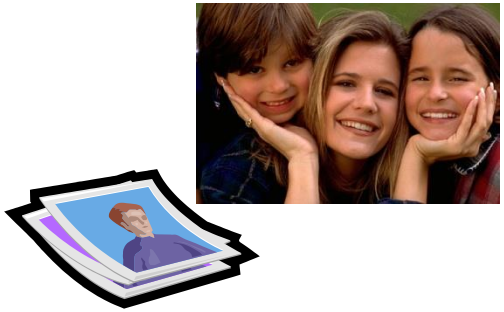
Abstract Classes vs. Interfaces

- Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation.
- If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.
- Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way.
- Abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).
- A class that implements an interface must implement *all* of the interface's methods. It is possible to define a class that does not implement all of the interface methods, provided that the class is declared to be abstract.

What is a Package?

- A package is a namespace that organizes a set of related classes and interfaces.
- Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another.
- **Definition:** A *package* is a grouping of related types providing access protection and name space management.

Simple Example

```
// file ClassOne.java in the directory
//  /home/s111111/java/Ex08/demopackage
package demopackage;
public class  ClassOne {
   public void methodClassOne() {
      System.out.println("methodClassOne");
   }
}
// file ClassTwo.java in the directory
//  /home/s111111/java/Ex08/demopackage
package demopackage;
public class  ClassTwo {
   public void methodClassTwo() {
      System.out.println("methodClassTwo");
   }
}
```

- Compilation:

  javac *.java

```
// file UsageDemoPackage.java in
// the directory
//  /home/s111111/java/Ex08/
import demopackage.*;
class UsageDemoPackage {
  public static void main(String[] args) {
   ClassOne v1 = new ClassOne();
   ClassTwo v2 = new ClassTwo();
   v1.methodClassOne();
   v2.methodClassTwo();
  }
}
```

- Compilation:

  javac UsageDemoPackage.java
- Run:

  java UsageDemoPackage

Referring to a Package Member

- Import the package member.

```
// importing the member
import demopackage.ClassOne;
. . .
ClassOne v1 = new ClassOne();
…
```

- Import the whole package.

```
// importing the whole package
import demopackage.*;
…
  ClassOne v1 = new ClassOne();
  ClassTwo v2 = new ClassTwo();
…
```

- Refer to the member by its fully qualified name.

```
…
demopackage.ClassOne v1 =
            new demopackage.ClassOne();
…
```

The Catch or Specify Requirement

- Code that might throw certain exceptions must be enclosed by either of the following:
  - A try statement that catches the exception – caching and handling exception
  - A method that specifies that it can throw the exception. The method must provide a throws clause – Specifying the Exceptions Thrown by a Method
- Three Kinds of Exceptions
  - checked exception: can anticipate and recover the exception. Checked exceptions are subject to the Catch or Specify Requirement (CSR). All exceptions are checked exceptions, except for *Error*, *RuntimeException*, and their subclasses.
  - error (unchecked exception) : cannot anticipate and recover, not subject to CSR, ex) system malfunction
  - runtime exception (unchecked exception): cannot anticipate and recover, not subject to CSR, ex) logic error or improper use of an API

Catching and Handling Exceptions

The try, catch, and finally block

```
try {
  // try block
}
catch (ExceptionType1 param1) {
  // Exception Block
}
catch (ExceptionType2 param2) {
  // Exception Block
}
 ……
catch (ExceptionTypeN paramN) {
  // Exception Block
}
finally {
  // finally Block
}
```

Statements that have some possibilities to generate exception(s).

Execute statements here when the corresponding exception occurred.

Do always

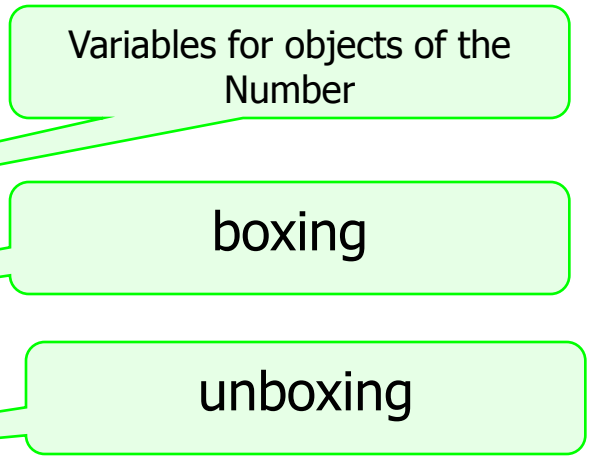
If a finally clause is present with a try, its code is executed after all other processing in the try is complete. It allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.

The throws Clause

- The checked exceptions that a method throws are as important as the type of value it returns. Both must be declared.
- If you invoke a method that lists a checked exception in its throws clause, you have three choices:
  - Catch the exception and handle it.
  - Catch the exception and map it into one of your exceptions by throwing an exception of a type declared in your own throws clause.
  - Declare the exception in your throws clause and let the exception pass through your method (although you might have a finally clause that cleans up first) .
- Throws clauses and Method Overriding: An overriding or implementing method is not allowed to declare more checked exceptions in the throws clause than the inherited method does.

Numbers

- **The Number class in java.lang**
  - Primitive types
    int i = 500;
    float gpa = 3.65;
    byte mask = 0xff;
  - Boxing: primitive type → object
  - Unboxing: object → primitive type
  - Example of boxing and unboxing
    Integer x, y;
    x = 12;
    y = 15;
    System.out.println(x+y);
  - There are corresponding Number class for primitive types
  - For example
    int --- Integer, char --- Char, double --- Double, long --- Long



The String Class

◆ String Literals, Equivalence and Interning

What's difference?
1) String str = "SomeString";
2) String str = new String("SomeString");
And String.equals() method

1) String Literal
2) Runtime String instantiation

It compares one object reference (str) to another… How about the next?

```
String str =
"SomeString";
if (str == "SomeString")
  answer(str);
```

Same reference!
* Auto intern

```
String str = new
String("SomeString");
if (str == "SomeString")
  answer(str);
```

Same reference?
* Not auto intern

- The intern() method of the String class returns a canonical representation for the string object.
- It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

```
int putIn(String key) {
  String unique = key.intern();
  int i;
  // see if it's in the table already
  for(i = 0; i <tableSize; i++)
    if(table[i] == unique)  return i;
  // it's not there – add it in
  table[i] = unique;
  tableSize++;
  return i;
}
```

All the strings stored in the table array are the result of an intern invocation.

The table is searched for a string that was the result of an intern invocation on another string that had the same contents as the key. If this string is found, the search is finished. If not, we add the unique representative of the key at the end. -> much faster

Manipulating Characters in a String

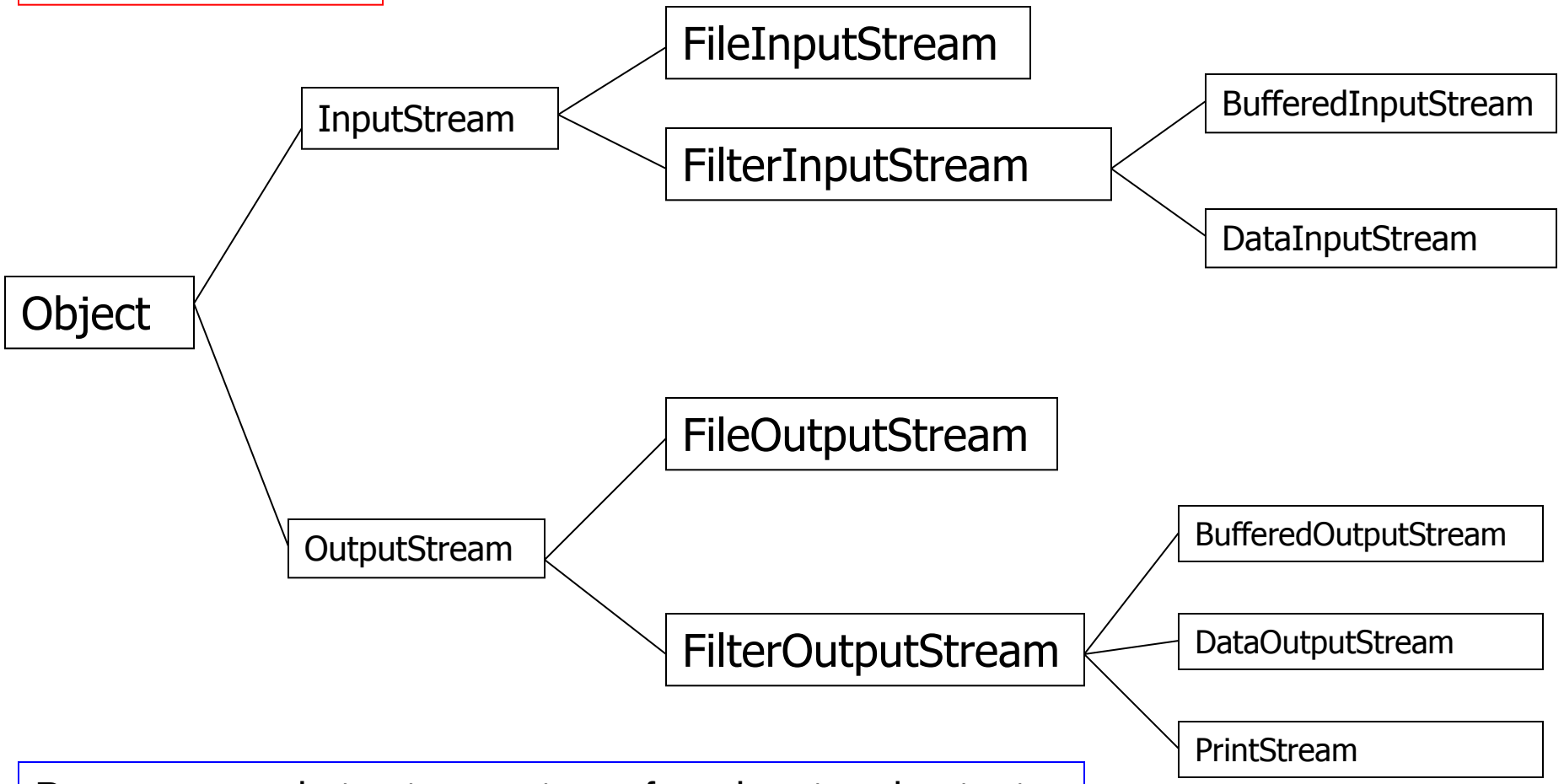
- **Other Methods for Manipulating Strings**
  - String[] split(String regex)
  - String[] split(String regex, int limit)
  - CharSequence subSequence(int beginIndex, int endIndex)
  - String trim()
  - String toLowerCase()
  - String toUpperCase()
- **Searching for Characters and Substrings in a String**
  - int indexOf(int ch)
  - int lastIndexOf(int ch)
  - int indexOf(int ch, int fromIndex)
  - int lastIndexOf(int ch, int fromIndex)
  - int indexOf(String str)
  - int lastIndexOf(String str)
  - int indexOf(String str, int fromIndex)
  - int lastIndexOf(String str, int fromIndex)
  - boolean contains(CharSequence s)
- **Replacing Characters and Substrings in a String**
  - String replace(char oldChar, char newChar)
  - String replace(CharSequence target, CharSequence replacement)
  - String replaceAll(String regex, String replacement)
  - String replaceFirst(String regex, String replacement)

Streams Overview

- Two major parts in the java.io package: character (16-bit UTF-16 characters) streams and byte (8 bits) streams
- I/O is either text-based or data-based (binary)
- Input streams or output streams → byte stream
- Readers or Writers → character streams
- Five group of classes and interfaces in java.io
  - The general classes for building different types of byte and character streams
  - A range of classes that define various types of streams – filtered, piped, and some specific instances of streams
  - The data stream classes and interfaces for reading and writing primitive values and strings
  - For Interacting with files
  - For object serialization

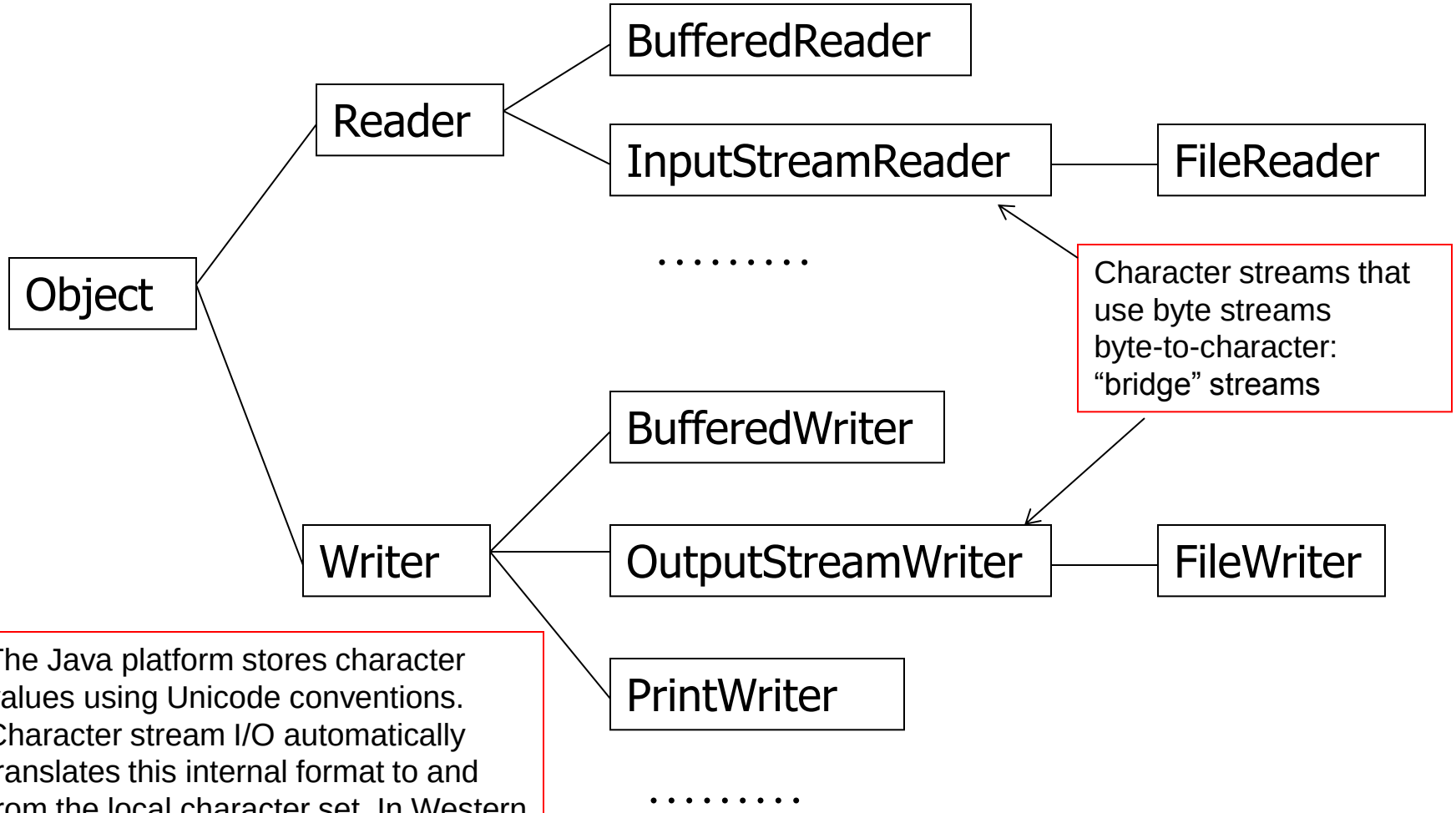
Byte Streams (Binary Streams)

For low-level I/O



Programs use byte streams to perform input and output of 8-bit bytes (octets).

Character Streams



The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

Filter Streams

```
import java.io.*;

public class UppercaseConvertor extends
      FilterReader {
  public UppercaseConvertor(Reader in) {
    super(in);
  }

  public int read() throws IOException {
    int c = super.read();
    return (c==-1 ? c :
      Character.toUpperCase((char)c));
  }

  public int read(char[] buf, int offset, int count)
    throws IOException
  {
    int nread = super.read(buf, offset, count);
    int last = offset + nread;
    for (int i = offset; i < last; i++)
      buf[i] = Character.toUpperCase(buf[i]);
    return nread;
  }
```

abstract class

```
public static void main(String[] args)
    throws IOException
  {
    StringReader src = new StringReader(args[0]);
    FilterReader f = new UppercaseConvertor(src);
    int c;
    while ( (c=f.read()) != -1)
      System.out.print((char)c);
    System.out.println();
  }
}
```

Function of the read() method was changed with filtering.

Overloaded read method: for reading in buffer which is array of character.

Filter streams help to chain streams to produce composite streams of greater utility. They get their power from the ability to filter-process what they read or write, transforming the data in some way.

Run:
% java UppercaseConvertor “no lowercase”
Result:
NO LOWERCASE

Conclusion

- In the Java Programming 1 course, we considered the key concepts of object oriented programming.
- The material provides good fundamentals for your further study.