# Java Micro-Tour
# (**draft 2011-08)

[rob rucker]

## Micro-Tour

This note is a quick tour of a few important features of the Java language. I have used material from "The Java Programming Language, 3rd & 4th editions" as a basis for these notes, as well as other books and my personal programming experience. Take a look at "Tour.java - source code illustration of Java Tour concepts" on page 8 and subsequent figures/diagrams for examples of these features.

## The Platform Context

The Java platform consists of (at least) the Java language, the Java class libraries, and the Java Virtual Machine (JVM). To accommodate all the levels of complexity involved in developing *software intensive systems*, the general term 'platform' can be considered as being made up of four interlocking platform components.

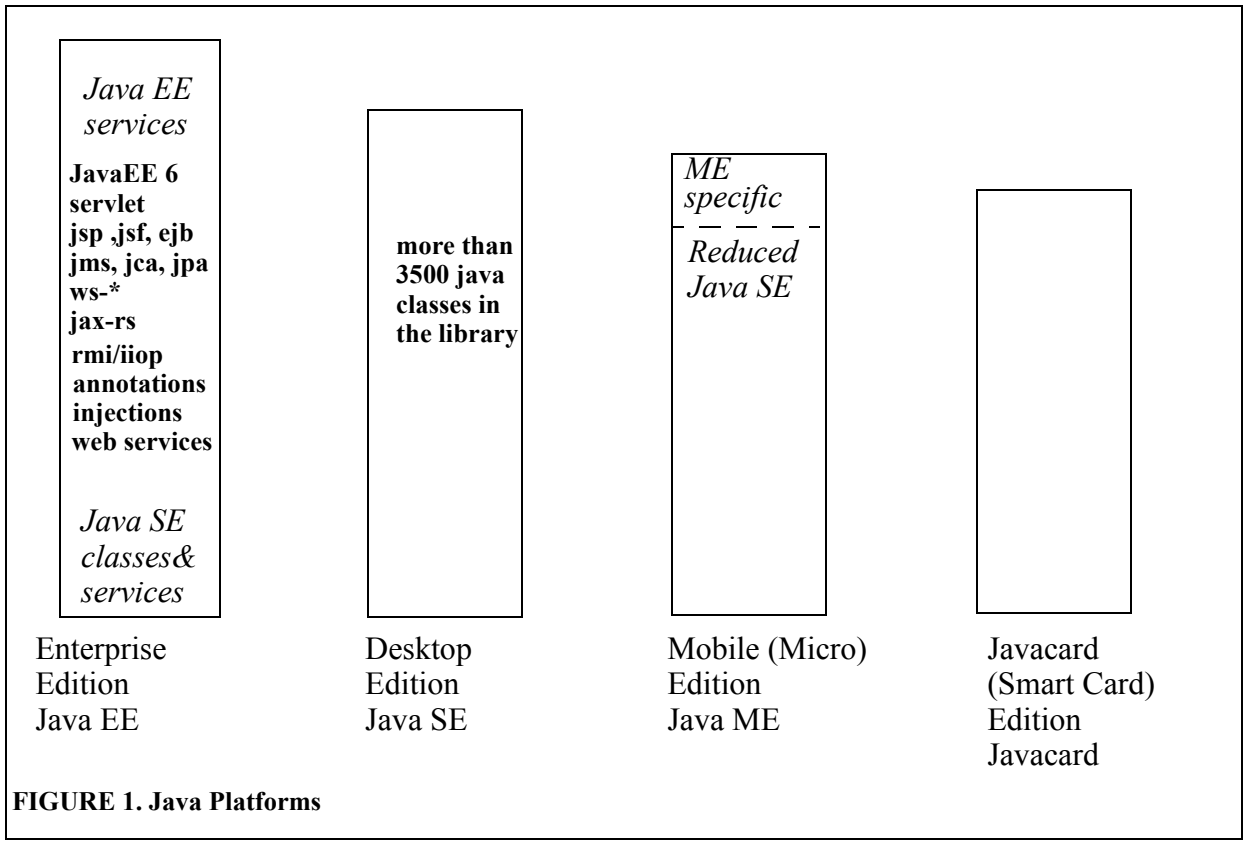



**FIGURE 1. Java Platforms**

Each of these component platforms has its own version of the class libraries as well as variations of the JVM.

## The Enterprise Edition Components (Tiered Applications)

The Enterprise edition is just that, designed for the most comprehensive tasks needed for a whole organization. What follows is a list of what kinds of components it makes available to the developer. The plan is to give support for large-scale scalable, secure, s/w intensive applications. The Java EE platform provides a framework and many services to help with the development activities, tasks, and techniques.

The word tiers means that functionality is separated (as much as possible) into areas that are called tiers. Usually such tiers are labeled as the:

### Client tier

This tier is made up of client programs requesting services from the web tier or directly from the middle tier. If the request is to the web tier then the request is handled by a 'server' that directs the request as necessary to other tiers and constructs a response that is sent back to the client.

Clients can be of many types, such as mobile devices, other computer programs, or even other servers. Clients usually run on different machines.

### Web tier

There is often one of these that mediates between the client and the business tier. Here is where a 'web server' operates that controls access to and from the middle tier. The primary tasks of the web tier are:

- Dynamically generate content for the client
- Get client requests, process them or redirect them, and collect and transmit responses back to the client.
- Manage the screens or pages that the client sees or interacts with.
- Keep the state of the session (e.g. think of the shopping cart applications where you want the system to remember what you earlier selected).
- Do some logic (not too much since that would inhibit separation of functionality)

There are specific technologies used within the web tier: Servlets, Java Server Pages, and JavaServer Faces technology.

Servlets are the 'Controller' classes that dynamically get requests, and construct responses, consisting of HTML pages (usually).

JavaServer Pages (JSP) are the 'View' documents that consist of Java code embedded within HTML pages. These documents are ultimately compiled into servlets for efficiency.

JavaServer Faces is a framework that is a layer above the servlets and JSPs, taking care of much of the low level tasks needed to build a web application.

### Middle (Business) tier (business logic resides here)

This tier often is responsible for the algorithms that represent business process and are responsible for storage and retrieval from persistent storage in the EIS. The core components in the business tier are:

Enterprise Java Beans (EJBs)

JAX-WS web service endpoints

Java Persistence API entities

**Enterprise Information tier (EIS)**

Often called the data tier. This layer holds the database servers, ERP, CRM, SFA, as well other legacy systems linked by Java Connector technology.

Components in the EIS layer are:

The Java Database Connectivity API (JDBC)

Java Persistence

J2EE Connector Architecture

Java Transaction API (JTA).

## Java EE Servers

A Java server is a server application that implements the Java EE platform API set as well as provides services. The Java EE servers correspond to the tiers discussed earlier. The Java EE servers provide services to the tier components in the form of a *container*.

*The Web Container*

This is the interface between web components and the web server. A web component is servlet, a JSP, or a JavaServer Faces page. The contain manages the life cycle of the various components.

*The Application Client Container*

The application client container is the interface between the client and the Java EE server. The application client container runs on the client machine and interfaces between the Java EE server and the components that the client invokes.

*The EJB Container*

The EJB container is the interface between enterprise javabeans and the Java EE server. The EJB container runs on the Java EE server and manages the applications EJBs.

## The Business Context for the Java Language

Think of the Java language as the programming 'low-level' implementation of the *services* that make up identified business processes. The language itself is part of a broader architecture called *The Java Platform.* The Java platform incorporates the language and supplements it with the Java class libraries (the APIs), and the Java Virtual Machine (JVM). You can specialize the Java Platform to manifest/implement various architectures and their associated processes such as: smart card applications, mobile and small device applications, real-time software applications, enterprise applications, distributed software, as well as peer-to-peer models. In general, any *software intensive* project can be approached using this Java Platform and its 'sub-platforms'. The latest comprehensive implementation involves 'service' principles resulting in a Service Oriented Enterprise (SOE).

(I think of a software intensive suite as one involving and correlating hardware, software, peopleware, paperware, networks, governance, as well as internal and external stakeholders).

# A Few Key Concepts

*Java Suite* - Think of a Java Suite as the computing component solution to a specified *software intensive* requirement. The suite itself is composed of interacting *programs* as noted below.

*Java Program* - Think of Java programs as a *community of responsible, cooperating classes and objects* that communicate by calling methods on each other. Each class, if well designed, is responsible for carrying out a set of responsibilities, called the *class contract*. Users of a class, (*who may extend it*) are entitled to count on the class fulfilling its contract. [This basic idea came from Rebecca Wirfs-Brock]

*Programs* - Java programs are built from classes and instances of classes, where the instances are called *objects*.

*Class* - think of a class as a factory plus the blueprints and instructions on how to build a particular kind of product. That product is called an *object* or is also called an *instance* of that class. From the class definition (which lays out the blueprints, instructions, and responsibilities), you can construct any number of these objects. Keep in mind, that, just like a factory, the class itself can have members apart from the products it constructs (these class-level (factory) members are designated as "*static*", see the section on static members ****).

*Object* - think of an object as a product constructed from the class definition introduced above. This object will have 'fine' structure consisting of members, such as variable, methods, plus possibly other nested classes and interfaces, as specified in the containing class. The object will also take up space in memory (on the 'heap') that depends on the types and sizes of its members.

In a more physical way, think of an object as the unique space the compiler generates as a "typed region of memory". The 'type' of memory generated corresponds to the class, and that region of memory is sufficient to hold all of the members of the object. That partition of memory dedicated to holding classes and instances of classes is called the *heap*.

*Members* - members of a class/object consist of fields, methods, and possibly other contained classes and interfaces. Yes, a class can contain other classes as members and you will later come to see that this is extremely helpful for expressing very tightly coupled logical concepts in the context of an encompassing class. (For example, in the Java Event Model, these nested classes are used extensively as 'Listeners').

*Fields* - these are data variables that belong to either the class or the object and outside of any methods. They are what the class/object "knows" or what is called the "state" of the class/object. The only types of fields are *primitive* field variables and *reference* field variables.

*Methods* - these are collections of statements using the Java language, that operate on other members of the class/object to manipulate their state/operations. These are the *operations* of a class. The operation of the methods make up the behavior of the class/object. The methods determine how the class/object "behaves". Other elements of a *Java Suite*, may invoke these methods to carry out needed interactions. To invoke a method, use the *object reference* of the target object followed by a 'dot', followed by the desired method. Arguments are passed to a method as a comma separated list enclosed by parentheses. (Even a method with no arguments still requires the empty open and closed parentheses).

*Interfaces* -"A Pure Design Intent Mechanism". As a Java program designer, you may want to insure that a certain method must be implemented by any class that interfaces with your designated class. You don't want to specify the details of the method, but only declare that it must be present as part of the contract. To do this, you specify the method signature, but not its body. This mechanism of interfaces provides a kind of *multiple inheritance*, but not the type that involves inheritance of implementation, just inheritance of type.

*Packages* - packages of Java classes and objects, are units of design that group together related classes and objects, and provide a unique namespace that avoids naming conflicts. (Remember,

your programs may be shipped all over the world and there is bound to be name conflicts unless precautions are taken). Think of how we name people, using a last name, first name, and if necessary, middle initial. The last name is the *package* name. On a broader view, we characterize someone as a North-American, South-east Asian, African, and so on. Here, the packaging is at a much higher level, but still is intended to group like entities together.

*Access Control* - To avoid allowing other classes and objects to corrupt your classes and objects, you specify *access control*. Access means the following:

All members of a class are always available to code in that class. Access from other classes is controlled as follows, in order of accessibility:

1. *public* - members are accessible wherever the class is accessible.
2. *protected* - members declared protected are accessible from subclasses of that class, from classes in the same package and or course, from within the class itself.
3. 'package' - This is the *default* access modifier and is not explicitly written out. Members declared with NO access modifier are accessible to classes in the same package and of course, within the class itself.
4. *private* - members declared private are only accessible within their class.

*Comments* - as a support for *Literate Programming* [Donald Knuth] there are three kinds of comments you can use:

/* these are the delimiters for regular C, C++, Java, C# comments that can span multiple lines of code */

// this is the delimiter that tells the compiler to ignore the rest of the line. Note that this comment doesn't span lines.

/** These are the delimiters for Javadoc comments that are extracted and converted into linked HTML pages */

This last type of comment forms the basis for production level documentation. I have used this option to auto-generate documentation that was acceptable (and billable!) to the client. I routinely document my code using this capability as you will see in the included file reproductions.

## Primitive and Reference Variables

To learn a language you will need to understand what kind of variables it supports and how they can be manipulated. For Java, the variables break down into two categories, primitives, and references. A "coffee cup" analogy really helps here, with ideas based on those of the authors K Sierra and B. Bates in their light-hearted but very insightful book, *Head First Java Programming* 2nd ed. (2005). Think of a variable as a *cup* which holds some value. As you know, cups have different sizes and that matches different variable types that hold different sized values.

### Primitive Variables are Cups or Containers that hold bit values

In the diagram below I show each primitive variable as a container/cup holding different sizes of values. The byte-sized cup holds 8-bit values, ranging from -128 to +127, while the long-sized cup holds 64-bit values. The sizes in these cases correspond to *data types*.

So, the size of the cup corresponds to the type of variable it represents, while the contents of the cup is the actual value, held as a bit pattern. For example, the byte type cup holds 8-bit patterns. For example, one particular value that the cup/variable might hold could be, say, decimal 25. The

contents of the cup in binary would be *00011001*. That is shown as 'pouring' into the byte cup in the diagram below. Below the cup picture I show a typical declaration and assignment for that variable. Notice from this analogy that trying to pour a 64 bit pattern into an 8-bit cup results in *overflow* and a loss of precision. So, be careful!

Identifiers for variables figure in here as well. When I go to my favorite coffee shop I ask for a certain size of coffee (cup size) as well as the amount of coffee as its content. Further, my name is usually written on the cup. That is its identifier. The name on the byte cup, for example, could be **by,** while the name/identifier on the float cup could be just **x**. (As a detail, note that I have to tell the compiler what size of variable is being declared in the case where I want to literally specify a *float* or a *long)*. That is indicated by the suffixes of **F** and **L** respectively. Otherwise the compiler assumes *double* and *int*.

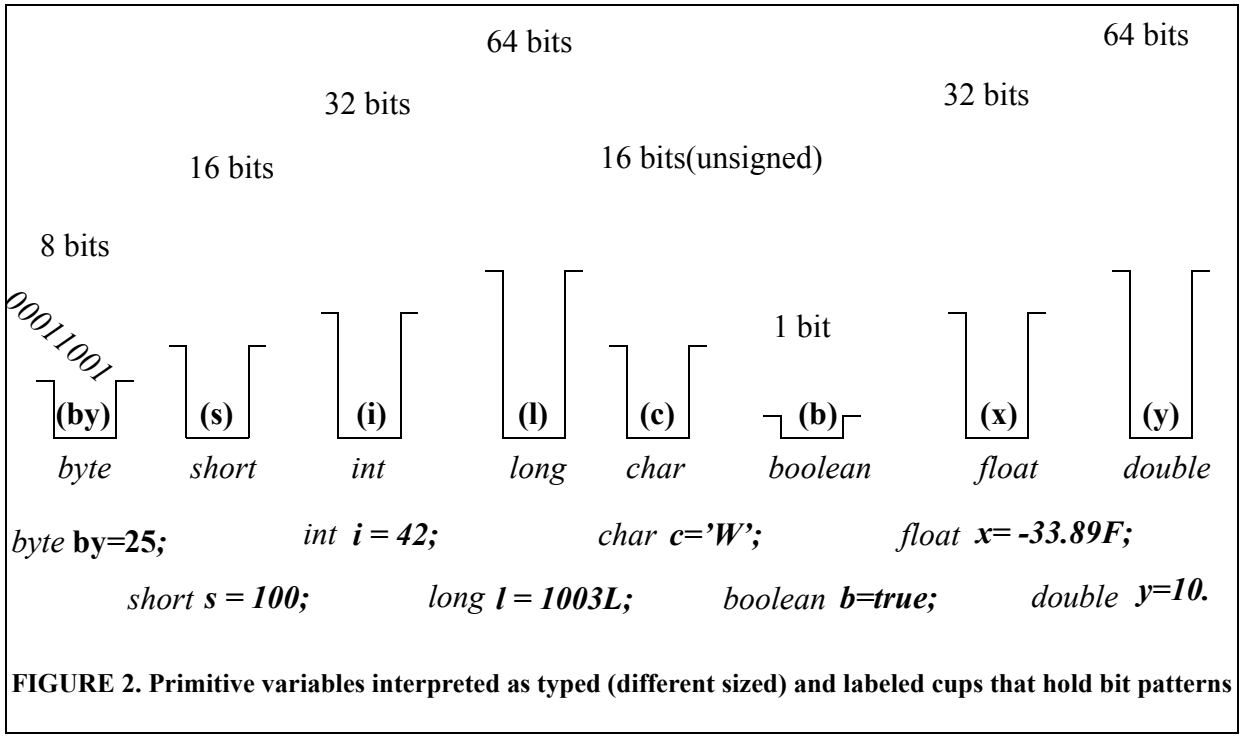



**FIGURE 2. Primitive variables interpreted as typed (different sized) and labeled cups that hold bit patterns**

**Reference Cups**

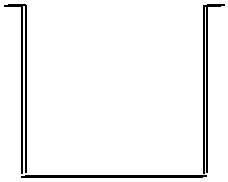
When it comes to reference variables the cup analogy still holds and is usefully enriched. In the case of reference variables, it helps to think of their cup size being constant for all reference variables. Even though this size is compiler dependent, I'm going to think of them all as being *128 bits*. The important feature of reference variables though, is, whatever their cup size, is the *rich internal structure* of their contained bit patterns that will be of interest. Just as primitive variables contain bit patterns representing actual values, like, 25 or -33.89, the bit patterns held by reference variable cups hold patterns that we can profitably think of as constructing a *remote control*. (Please suspend disbelief for a moment as I develop this analogy)!

*Remote controllers built from bit patterns residing in a cup that contains 128 bits.*

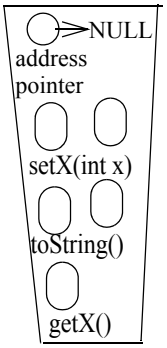
I am going to work through some ideas that I hope will help you think about reference variables, their construction and usage and their relation to *objects*. Take a look around your home or apartment and you might see a number of remotes lying about. Some are programmed to manipulate

DVDs, some CD players, some for TVs and so on. There might even be more than one DVD remote since you might have bought more than one *new* DVD player over time, as I have. These DVD remotes represent a type of remote that I will call *DVD*s To tell my DVD remotes apart, I put a label on them and write an identifier, say, dvd1, dvd2, and so on. Each time I buy a *new* DVD player I get another remote that I treat similarly. Notice also that there are a number of buttons on my remote that, when pressed or switched, result in a change of behavior of the particular DVD it is linked to. Now let me relate this back to the Java world.

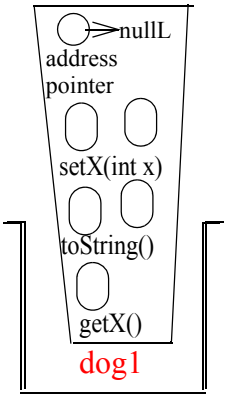
**Note that a container holds the value of a variable. If it is a primitive container then it holds the literal value. If a reference container, then it holds a reference variable.**



A reference cup holds 128 bits. (compare with 32 bits to hold an integer).




A reference variable is composed of 128 bits that form a pattern (a **remote control)**. This pattern (remote) is of a certain type, such as 'Dog' or 'ComplexNumber"




The name of the reference variable is dog1 and is a consequence of specifying:
Dog dog1 = null;
note that the address points to 'null' (nowhere!)

**FIGURE 3. Reference variables and their containers**

TBD !

## A Java Source class File: Tour.java

```
1  /* Tour.java */
2  package ch10;//'package'is a grouping designator for a set of Java classes/objects
3  public class Tour           //The public declaration of the Tour class
4  {                           // A curley brace starts the class body
5    static private int MAX=5; //class level field,independent of any object.
6    private int cities = 4;  //object level field, with private access only.
7    public int getCities()   // Object level method, with public access.
8    {                        // Start method body.The method is intended to return
9     return cities;          //  the  current value of cities
10   }//end getCities         //curley brace designates end of a method
11   /** Start a Javadoc comment here --
12    * This main method is used to illustrate object construction
13    * @param  args is a String  array that will hold command line input Strings
14    */                     //end the  Javadoc comment here
15   public static void main (String[] args)//class level method, independent of
16                                          //any object
17   {                                      //start method body
18    Tour t = new Tour(); //construct an object of type Tour
19    int temporary = t. getCities(); //return  the value of cities and
20                                    // assign  it to the temporary variable
21    System.out.println("Tour of Cities @: "+ temporary);// print to console
22   }//end main
23 }//end class Tour
```

**FIGURE 4. Tour.java - source code illustration of Java Tour concepts**

### A Line by Line Explanation of Tour.java

I have pretty much explained this class by means of the three types of embedded comments (see if you can identify them as well). A couple of more words might help; though.

Line 18: This line constructs a new object on the heap, a Tour type object, that is loaded with the values of the class members. After this construction, the new object and its members, are accessible through the reference variable "t".

Line 19: The object constructed in line 18 is accessed via its reference variable "t" and one of its methods, "getCities()" is invoked. (Remember that a Java program is a *community of cooperating objects and classes*, communicating via these kinds of method invocations). After the invoked method runs, it returns to the calling program an integer value that is then assigned to a temporary variable I call `temporary`.

Line 21: this line outputs both literal text (Tour of Cities @:) and text resulting from the conversion of the temporary variable to a string. This last operation happens automatically. The "+" is the string concatenation operator in Java.

# Methods and their modifiers

The methods of a class usually manipulate the fields of the class.

The structure of a method consists of: a *method header* and the *method body*.

The method header: consists of an optional set of *modifiers*, an optional set of *type parameters*, the method *return type, the signature,* and an optional *throws* clause.

*Signature*: this consists of the method name and the parameter type list (enclosed in parenthesis). The parameter list can be empty

Method modifiers

- annotations
- access modifiers
- abstract
- static
- final
- synchronized
- native
- strict floating point

*Type parameters*

This refers to the new innovation in Java called *generics*. This option is part of the intent to make Java programs as type safe as possible. Generics is one more way to do this. Illustrations of Generics in actions will be shown in the Collections section (TBD).

**Static methods**

Static methods, are also called *class methods*, are associated with the class itself, that is, the template. You can use these methods just by writing in the class name 'dotted' with a static method. For example, *Math.random()*, returns a random number by using the class name 'Math" and a static method '*random()*'. These static methods perform services for the class as a whole. Note that a static method can access only static fields since a class level method doesn't have any object instance to refer to, that is to say, it has no *this*!

**Method Notes in General**

When method is invoked, the caller must provide the proper *argument* types that match the defined *parameter* types. The compiler will do some type conversions for you via implicit conversions and the Boxing and UnBoxing features of the language that *wrap* and *unwrap* primitives from their object wrappers.

Every method has a return type of type primitive, reference, or void. The void is used to indicate that the method does not return any value.

A method can have a variable number of arguments (see System.out.printf(. . . ) ). This works when you specify the last parameter in a parameter list as a sequence of a given type. The sequence being indicated by an ellipsis. These methods are known as variable argument or *varargs* methods.

public static void print (String . . . names); // this static method takes a variable number of Strings as its argument.

**Overloading and Overriding**

*overloading*

Remember the idea of a method *signature*, which consists of the method name and its parameter list? Well, you are allowed to have two or methods methods with the *same name*, if their parameter

lists are distinguishable (by the compiler). Notice that this is irrespective of return types or exception that might be thrown. Those features are not used to resolve differences. Usually, you will want to overload methods where the same information is presented, but in a different format. This again is usually for the convenience of the user of your software. The three signatures below with the same name of 'process', are different (from the compilers' perspective) and thus distinguishable, and thus overloaded.

public void process (String name, int processID);

public void process (String name, int processID, String priority);

public void process (String name, String priority, int processID);

*overriding*

This happens when you replace the superclass's implementation of a method with your own implementation. The signatures must be the same, name and parameter list must match, *and, the return type* must follow special rules. If the superclass's method has a *reference* return type, then the overriding method can declare a return type that is that type or a subtype. If the return type of the super class is a primitive, then you must match it exactly.

Access - a subclass can only provide more or the same access as the super class, not less. That is, if the super class access is package level, then the overriding method can have package level access or public access.

## Constructors (Part I)

Classes can have constructors, which are special blocks of statements intended to allow efficient initialization of objects. Often you will want to set up new objects to have specific values for its fields. While you can use getters and setters, it is very convenient to allow the user to decide on needed values at initiation time. Additionally, you might want to do some calculations as part of the new objects field values which you can't do with simple assignments.

A special syntax is used for constructors: they must have the same name as the class they are part of and can have any of the access modifiers that a class can. They are a lot like methods in that they can take zero or more parameters but have no return type. There is a constructor shown on line 18, the *Tour()* constructor.

### Constructing an A object

The constructor code is run before any access to the created object is allowed, and, the constructor code is run *after* the instance variables are set. That is, constructors are called after the instance variables have been assigned their default values and after initialization of variables is done.

One constructor can call another from the same class by using the keyword *this*.

Consider the class below and let me trace through the steps before and after constructor invocation.

```
public class A extends Object
{
      int a = 42;
      double ax;
public A()
      {
      ax = 2.718;

      }//end ctor
```

```
}//end class A
```

If I have some code that constructs an object of type A, such as:

```
A aRef = new A( );
```

Here is the sequence of operations. Note that A extends the top most Java class, *Object*. Every Java class is a child of *Object* but you don't need to explicitly write that since the compiler automatically 'knows' that.

**TABLE 1. Construction and Initialization**

| Step | What it does | value of a | value of ax |
|---|---|---|---|
| 0 | First, set all fields to their default values (no matter if they have initial values explicitly written in) | 0 | 0.0 |
| 1 | A's constructor is *invoked* (note, not *executed* yet since the A constructor will call *its* constructor. That is, it will call its super class constructor, which is Object() | 0 | 0.0 |
| 2 | top level Java *Object* constructor invoked and executed | 0 | 0.0 |
| 3 | A's field initialization are executed, as stated by you. | 42 | 0.0 |
| 4 | A's constructor is now *executed* | 42 | 2.718 |

# Extending Classes (Inheritance) and PolyMorphism

When a class is *extended* or *subclassed*, then an object of that extended class can be used whenever the original class can be used. This capability is called *polymorphism*. This means that an object of a given class can play multiple roles, either as an object of its own class or as an object of a class it extends. This supposes though, that the subclass honors the 'contract' expressed by its parent.

*What's a Contract?*

In modern design, the notion of a contract implies that an outside user of the entity, in this case a Java class, can depend on certain features and behavior.

Since a class consists of members such as methods and fields, the contract consists of how these members can be accessed and how they promise to behave. The contract is what the designer is obligated to do. Where is all this documented? Often these contracts only exist in the designers head, but, in modern software houses, there is a great emphasis on *Javadocs* and supplemental design documents. It should be obvious that there is a serious need for comprehensive and accurate documentation so that the *software intensive system* can be enhanced, or at least maintained.

### Inheritance

There are two forms of inheritance involving class extensions:

- inheritance of contract or type - this is where the subclass (extended class) receives the type of the superclass and so can be used polymorphically whenever the superclass could be used.
- inheritance of implementation - this is where the subclass (extended class) receives the implementation of the superclass's accessible fields and methods.

When you do class extension (subclassing) you always get both forms of inheritance. There is a way though to get just inheritance of type, and that is by using *interfaces*. Interfaces are a way to get multiple inheritance without the dangers that inheritance of implementation would bring.

# Constructors in Extended Classes (Part II)

When you declare that you extend a class from a class A, say the extended class is named B, then B gets (inherits) all the fields of its superclass A. When you actually construct a 'B' object, say **b**, its own fields must be correctly initialized as well as those of its superclass. The way it works is that when you construct a **B** type object, a superclass object must be constructed first as well as its' superclass object, as well as its' superclass object, . . . right up to the top of the Java class hierarchy, finally ending with an object **o** of the primordial *Object* class. In the case I just illustrated we would only need to go up from A to B to Object. See the discussion below for a detailed sequence of operations during construction.

### Constructing an Extended Object

Let me pictorially run through a typical sequence of operations that are triggered when you go to construct an object **b** from class B which extends class A, which (implicitly) extends Object.

Out on the heap, where these objects are physically constructed, a picture of the object b might look like a set of nested rectangles representing the (contained) constructed objects. That is, **o** is constructed first, followed by **a**, followed by **b**. At the center is the object **o**, constructed from the top most class, Object. All of these can be considered as being contained by object **b.**

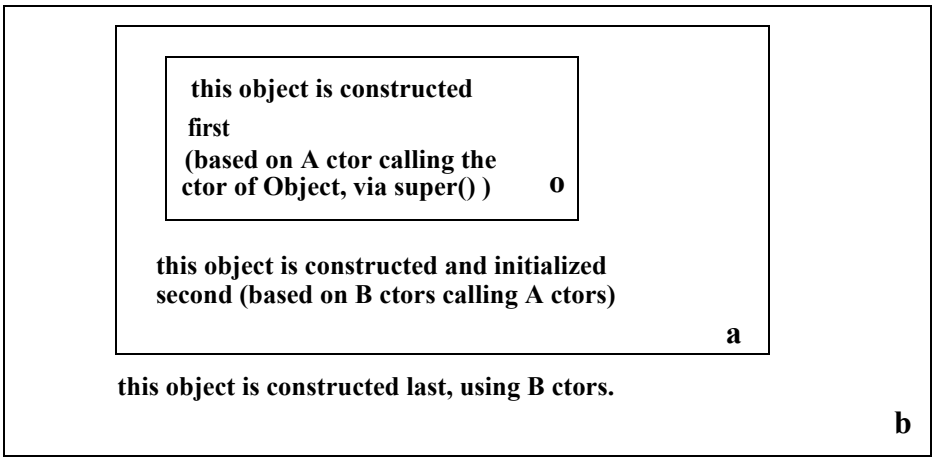



**FIGURE 5. Class B extends class A which extends the top class Object: o, a, and b are constructed objects.**

### Construction and Initialization for Hierarchies

Let me do a detailed run through of what happens when you make an object at the bottom of a hierarchy. Let me repeat class A here and also introduce class B, which extends A.

```
public class A extends Object
{
      int a = 42;
      double ax;
public A()
      {
      ax = 2.718;
      }//end ctor
}//end class A
public class B extends A
```

```
{
      int b = 777;
      double bx;
public B( )
      {
            bx = ax + 0.423;
      }//end ctor
}//end class B
```

**Making a B type object, steps and outcomes.**

Given the hierarchy of B extending A extending Object, what happens when I make an object of type B as in the code fragment below?

```
// previous code here
B bRef = new B( );
// more code here
```

**TABLE 2. Construction and Initialization of a B object**

| Step | What it does | value of a | value of ax | value of b | value of bx |
|---|---|---|---|---|---|
| 0 | First, set *all* fields in the hierarchy to their default values (no matter if they have initial values explicitly written in) | 0 | 0.0 | 0 | 0.0 |
| 1 | B's constructor is *invoked* (not executed yet since its superclass ctor is invoked) | | | | |
| 2 | A's constructor is *invoked* (note, not *executed* yet since the A constructor will call *its* constructor. That is, it will call its super class constructor, which is Object() | 0 | 0.0 | 0 | 0.0 |
| 2 | top level Java *Object* constructor *invoked* and then executed (fields here are opaque) | 0 | 0.0 | 0 | 0.0 |
| 3 | A's field initializations are executed | 42 | 0.0 | 0 | 0.0 |
| 4 | A's constructor is now *executed* | 42 | 2.718 | 0 | 0.0 |
| | B's fields are now initialized according to your instructions | 42 | 2.718 | 777 | 0.0 |
| | B's constructor is now *executed* | 42 | 2.718 | 777 | 3.141 |
| | | | | | |

## Constructor Order Dependencies

To make the above diagram and my explanations a little more formal, let me quote from the source.

According to Arnold and Gosling's *Java Programming Language* 4th edition, (p. 81), the constructor order dependencies go like this:

When an object is created, memory is allocated for *all* its fields, including those inherited from superclasses. Those inherited fields are set to default initial values, based on type (zero for all numeric types, null for all reference types, false for boolean and '\u0000' for char). Then, construction moves into three phases:

1. Invoke a superclass's constructor
2. Initialize the fields using their initializers and initialization blocks
3. Execute the body of the constructor

So, this sequences says: first the implicit or explicit superclass ctor is invoked. If there is a *this* explicitly used in a constructor however, then the chain of the *this* invocations is followed until an implicit or explicit superclass constructor invocation is found.

## Enum - A Special Kind of Class

"An enumeration type -- sometimes known as an enumerated type, or more simply as an *enum* -- is a type for which all values of the type are known when the type is defined. For example, an enum representing the suits in a deck of cards would have the possible values hearts, diamonds, clubs, and spades; an enum for the days of the week would have the values Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday. These are called *enum constants*. In some programming languages, enums are nothing more than a set of named integer values, but in the Java programming language an enum is a special kind of class.

"Each enum constant actually refers to an instance of the enum class." [Java Programming Language, 4th ed. pg. 151]

```
1  /* Book.java , a special class of type 'enum' */
2  package enums;//Group together a set of classes/objects under the name 'enums'
3  /**An enum class illustration. Each enum 'constant' is actually
4   * an instance of the enum class.So, each constant is like a 'little' object,
5   * complete with its own members.*/
6  public enum Book // An enum is a special kind of class
7  {                // Enum constants are written in upper case (by convention),
8                   // Note the 'fine structure' of each constant
9   SOA("Service Oriented Architecture", 2007),//here is the first 'enum' constant
0   JL("Java Programming Language", 2006),     // here is the second
1   HFJ("Head First Java", 2007);              //here is the last enum constant
2  private final String title; // an enum class can also contain regular variables
3  private final int year;    //these two will hold the enum constant's data
4  Book(String title, int year) //an enum constructor that loads the
5  { this.title = title;  //enum constants' internal variables
6    this.year = year;
7  }//end ctor
8  public String getTitle()
9  {return title;
0  }//end getTitle()
1  public int getYear()
2  {return year;
3  }//end getYear
4  }//**end enum Book
```

**FIGURE 6. An enum class with enum constants and instance constants**

The enum class is interesting when you want to save more detail about each enum constant (which, in reality, is an object itself). Notice that the constants are declared with two arguments. For example, the first enum constant SOA, is declared with the arguments “Service Oriented Architecture” and 2007. What is the system supposed to do with this information? Well, notice that there is a constructor, Book (String title, int year). So, when the enum constant SOA is built, that constructor takes the two arguments and inserts them into the object that represents SOA, exactly like any other constructor. The difference is that the constructor arguments are *right inside the class definition*!

# Generic Types

The biggest change in years to the language was the introduction of *Generics* in "Java 5" in 2005. In keeping with the idea that Java is a strongly typed language (every element of Java must have a type), this *generics* language feature extended the areas of type safety. An example is essential here! (see "Generics example of enforcing the type in a 'Dog' ArrayList (no Cats allowed!)" on page 17 below).

Java has an extensive library of collection classes that allow you to store, manipulate, and retrieve collections of classes and objects. The most important collection class is ArrayList, a type of List that grows or shrinks dynamically according to what you put in or take out.(This is in contrast with an array where once you specify a size, that stays fixed). Prior to Java 5, you could insert any type of object you wished into this List. This is not a good thing! What you want is to insure that only specific types (or their compatible types) are allowed into the list, preventing inadvertent mistakes. For example, only Dog references are allowed in the ArrayList below, no Cats need apply! So, in Java 5 you can now specify that you absolutely positively want only Dog references (or compatible types) to be allowed in the List. Look at the code below and see that I have set up an ArrayList<Dog>. I am telling the compiler to only allow Dog references (or compatible types) to be added to this ArrayList.

```
1  /* Generic */
2  package demo;
3  import java.util.ArrayList;// a special type of List
4  public class Generic
5  {
6   public static void main (String[] args)
7   {
8    ArrayList<Dog> aList = new ArrayList<Dog>();// hold only DOg references
9    aList.add(new Dog());//insert a Dog reference
10   aList.add(new Dog());
11   aList.add(new Dog());
     aList.add(new Cat());//try to insert a Cat reference **** Baaaaaad ****
13  }//end main
14 }//end class Generic
15 class Dog{}// a modest class definition!
16 class Cat{}//Note that I can define multiple classes in a file,
17  //but only 1 of those can be public,//
18 // which determines the file name, Generics in this case.
19 /*
20 Compiling 1 source file to C:\aaPrograms\nb6\JavaTour\build\classes
21 C:\aaPrograms\nb6\JavaTour\src\demo\Generic.java:12: cannot find symbol
22 symbol  : method add(demo.Cat)
23 location: class java.util.ArrayList<demo.Dog>
24   aList.add(new Cat());
25 1 error
26 BUILD FAILED (total time: 0 seconds)
```

**FIGURE 7. Generics example of enforcing the type in a 'Dog' ArrayList (no Cats allowed!)**

## Checked Exceptions

Java has a very neat way for you, as a programmer, to cope intelligently and efficiently, with errors in code. The down side is that you need to plan ahead as to what you are going to do if errors can potentially occur. In Java, you won't be able to ignore them (not *you* of course, but a colleague might!)

You have to specify, for each method you write, whether it *might* generate an error when someone calls it. The specification is in the form of a "throws" phrase, as shown below in the badMethod() method. That method declares that it can potentially throw an exception object. Your code must document this potential error in the header of the method, as is shown. That way, other users of your code can make their plans to 'catch' and deal with this potential error.

The exception is an *object* itself and is *caught* by code further back on the call stack. The catch part of the process allows the catching code to possibly recover from the error or at least exit gracefully.

The paradigm here is the *try-catch-finally* trio. This works out to:

1. first try to execute a method (or some other block of code).
2. If that code throws an exception, prepare for the exception object to be passed to another block of code, designated as a *catch* block.
3. Finally, clean up from the exception or the normal code path. A finally block of code always runs, exception or not. This is important when you want to release resources, regardless of previous outcomes.

```
1  package demo;
2  public class MethodException
3  {
4   public void badMethod()throws
5    BadMethodException
6   {
7     throw new BadMethodException();
8   }//end BadMethod
9     public static void main (String[] args)
10    {System.out.println("In BadMethod main() ");
11     MethodException bad = new MethodException();
12     try
13     {System.out.println("Just before 'badMethod() call");
14      bad.badMethod();
15      System.out.println("Just after 'badMethod() call");
16     }//end try
17     catch(BadMethodException ex)
18     {System.out.println("In BadMethodException catch block ");
19     }//end catch
20     finally
21     {System.out.println("In Finally, system  always does this block of code");
22     }//end finally
23    }//end main
24 }//end class MethodException
25 class BadMethodException extends Exception
26 {}//end class BadMethodException
```

# Input and Output (I/O) Services in Java - The java.io.* package

The key idea of input/output is that it proceeds by using ‘streams’. This is a concept that is used everywhere in the programming world for input and output of data. By treating all input and output as interfacing with streams, the programmer is shielded from all of the underlying operating system/network intricacies. For example, in “Byte Stream Input Read” on page 21, the programmer (me) only has to specify the file to read from, and all the details are taken care of.

An *input stream* is an ordered sequence of data (bytes or characters) connected to a source. The source could be, for example, a file, a network connection (socket), your program’s internal array of data, or your typed input at the console.

An *output stream* is an ordered sequence of bytes that is connected to a destination (target). The destinations can again be all of the types you saw for input. The combination of reading and writing to files is very common as is reading/writing data over the internet using sockets, which happens every time you type in a URL in your browser’s location field and click.

Both types of streams support the idea of being open or closed. You open a stream to write to it or read from it and close a stream when you no longer read or write and wish to release its resources (such as file connections).

> There are two general categories of I/O classes used by Java to manipulate streams. One category, called *connection stream classes*, has the capability to actually connect to sources and destinations. The second category, called the *chaining stream classes*, don’t have the capability to connect directly, but are chained to those classes that can. For example, a connection stream class would be FileInputStream that has as its argument the actual file it will connect to. Another class, BufferedInputStream is a chaining class that, when connected to FileInputStream, provides additional capabilities. BufferedInputStream can’t, by itself, connect to a source or destination and thus has to depend on a connection stream before it can provide services.

The *java.io* package has two major sets of interfaces and classes dedicated to reading and writing data streams. The first set is tailored to reading and writing 8-bit (byte) chunks of data, which is suitable for working with binary data such as image files. These are called *byte oriented* reads and writes.

The second set of interfaces and classes is focused on reading and writing 16-bit chunks which are interpreted as character data. Use these for reading and writing, for example, text files, or any other character data sets.

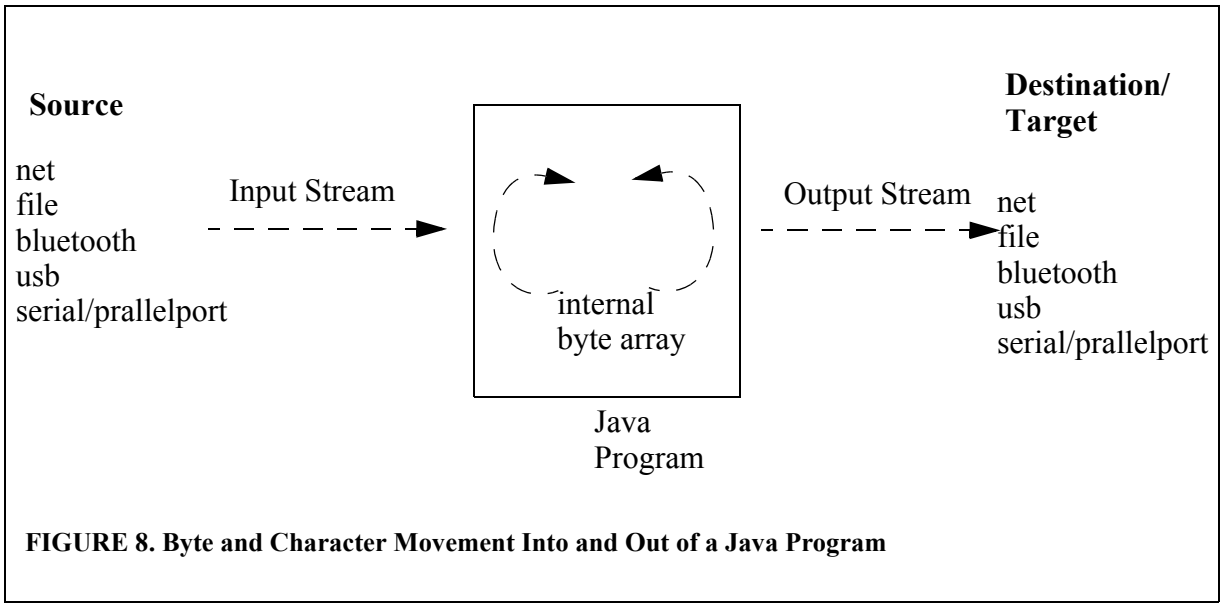



**FIGURE 8. Byte and Character Movement Into and Out of a Java Program**

## Byte Oriented Input and Output Interfaces and Classes

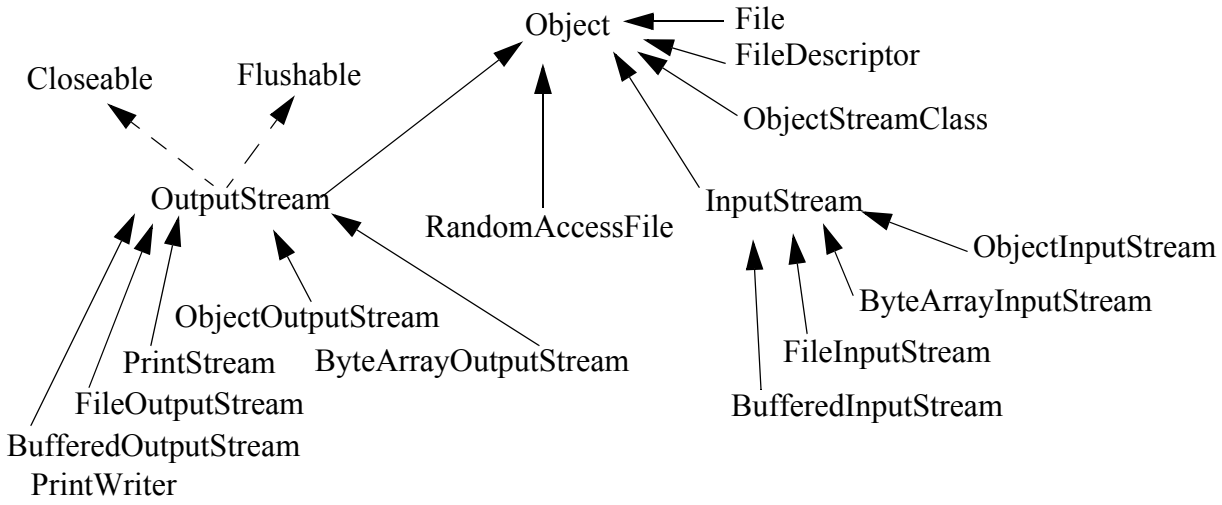
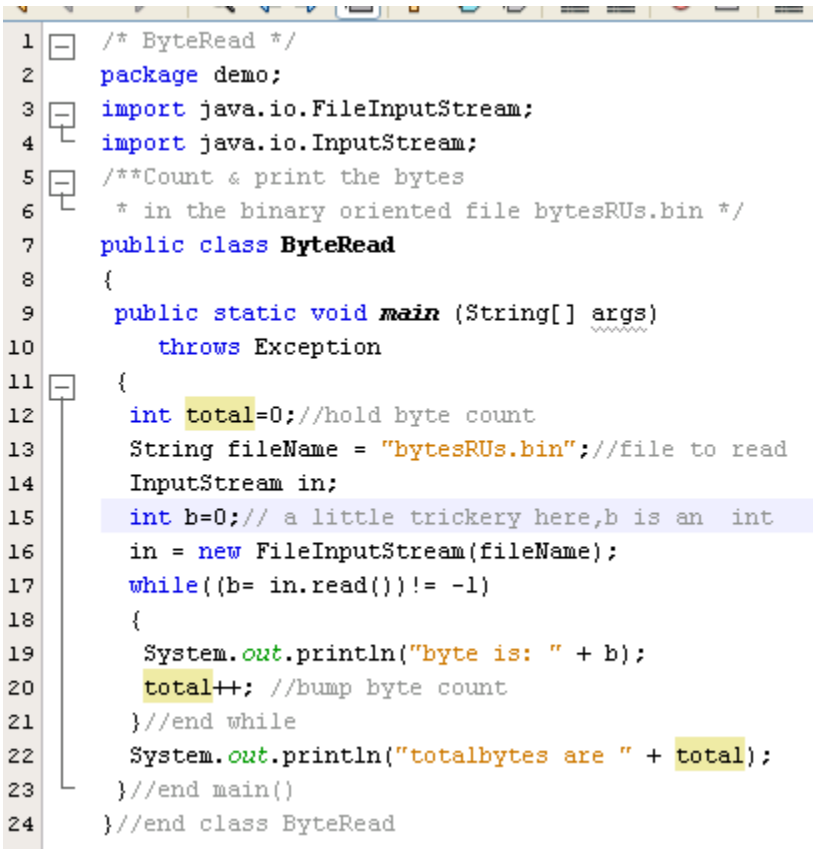



**FIGURE 9. (Partial) Byte Stream Type Tree**



```
1  /* ByteRead */
2  package demo;
3  import java.io.FileInputStream;
4  import java.io.InputStream;
5  /**Count & print the bytes
6   * in the binary oriented file bytesRUs.bin */
7  public class ByteRead
8  {
9   public static void main (String[] args)
10     throws Exception
11   {
12    int total=0;//hold byte count
13    String fileName = "bytesRUs.bin";//file to read
14    InputStream in;
15    int b=0;// a little trickery here,b is an  int
16    in = new FileInputStream(fileName);
17    while((b= in.read())!= -1)
18    {
19     System.out.println("byte is: " + b);
20     total++; //bump byte count
21    }//end while
22    System.out.println("totalbytes are " + total);
23   }//end main()
24  }//end class ByteRead
```

***** Input File *****
**BytesRUs.bin**

```
1  12345ABCab
2  0()!
```

***** Output to Console*****

byte is: 49
byte is: 50
byte is: 51
byte is: 52
byte is: 53
byte is: 65
byte is: 66
byte is: 67
byte is: 97
byte is: 98
byte is: 10 (linefeed)
byte is: 48
byte is: 40
byte is: 41
byte is: 33
totalbytes are 15

**FIGURE 10. Byte Stream Input Read**

## Character Streams

To read and write streams of characters you use another set of interfaces and classes, that mostly match up to the byte input and output interfaces and classes. The diagram below is a partial type tree showing the ones I have used the most.

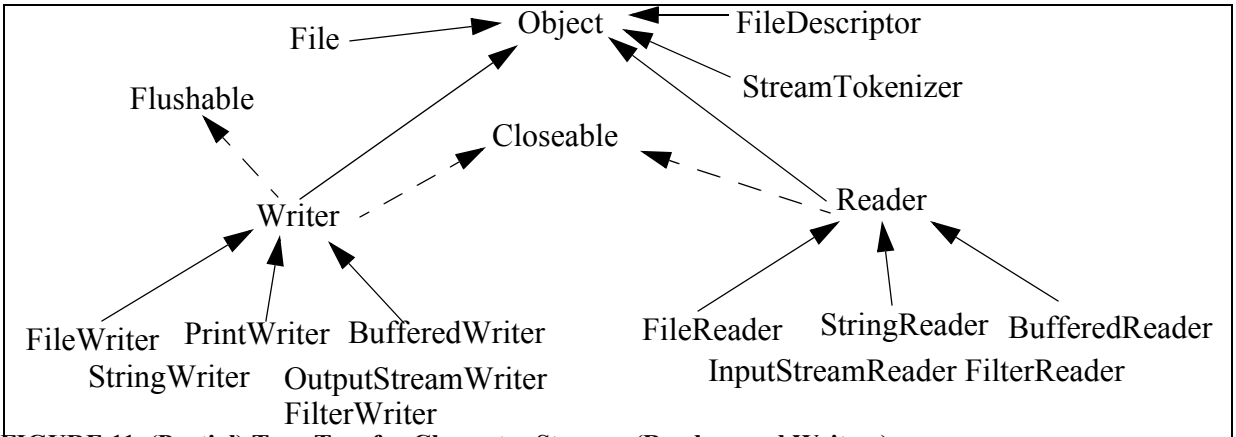



**FIGURE 11. (Partial) Type Tree for Character Streams (Readers and Writers)**

## Character Read From the BytesRUs.bin File

```
1  /* CharacterRead */
2  package demo;
3
4  import java.io.FileReader;
5  import java.io.Reader;
6
7  /** Show how to read characters from a file */
8
9  public class CharacterRead
10 {
11  public static void main (String[] args) throws Exception
12  {
13   String fileName = "BytesRUs.bin";
14   Reader in;
15   in = new FileReader(fileName);//establish connection
16   int character;//receiving variable
17   int total=0;//total characters
18   while( (character=in.read())!=-1)//read until End of File
19   {
20    System.out.println("Character = " + character);
21   }//end while
22  }//end main()
23 }//end class CharacterRead
```

**FIGURE 12. Character Stream FileReader Example**

# Annotations

```
1  /*ClassMetadata.java Annotation Type */
2  package demo;
3
4  /**Demonstrate an Annotation Type that will
5   * help to document data about a class,
6   * such as who
7   * created it, and when it was last revised.
8   *
9   * @author rob
10  */
11 public @interface ClassMetadata
12 {
13  String created();
14  int revision();
15 }//end annotation type
16
```

**FIGURE 13. Annotation Interface**

**Annotated Class**

```
1  /*ClassMetadata.java Annotation Type */
2  package demo;
3
4  /**Demonstrate an Annotation Type that will
5   * help to document data about a class,
6   * such as who
7   * created it, and when it was last revised.
8   *
9   * @author rob
10  */
11 public @interface ClassMetadata
12 {
13  String created();
14  int revision();
15 }//end annotation type
16
```

**FIGURE 14. An Annotated Class (using the ClassMetadata annotation type)**

# Threads

## Explore Threading in Java (rev 1 2011-08-24)

### Initial Ideas

(I have referenced the SCJP exam notes for some of this material as well as various other books)

In Java, a 'thread' has two different meanings:

1. An instance of the class Thread.java
2. A thread of execution (TOE). Each thread has its own local variables. program counter (pointer to current instruction being executed), and lifetime.

An instance of the class Thread is just a POJO object like any other object in Java with variables, methods, and lives on the heap (and dies there as well). A *thread of execution* though is something else, it means that I am talking about a *process*, a lightweight process that has it's own *call stack*. Think of a thread as a stack of executable statements, with one stack per thread.

For example, the *main()* method, that starts off your program suite, runs in one thread, the *main* thread (who would have guessed?). This main thread is spawned by the JVM, which begins execution with the main() method, executes all the statements in main(), and dies when the main() completes. Note that a second thread is also running, the Garbage Collector meaning that every Java program is by default, multi-threaded!

If you look at the main TOE call stack you will see the main() method on the bottom, the first method on the stack. When you create a new 'thread', a new stack is created so that when methods are called from that thread, they are in a separate call stack.

We say that the second new call stack(TOE) is running concurrently. For single processor systems this equates to allocating time slices to threads but for multicore processors, these can be actual concurrent processes.

**A thread in Java always begins as an instance of *java.lang.Thread.* That means that I can look into the *Thread* class and find methods for managing threads such as creating, starting, and pausing them. We will need to know about start(), yield(), sleep(), join(), synchronize(), and run().

Think of what you want to do, such as analyze a year of stock prices, as the *job* you want done. (Put the code that does the *job* inside the run() method). Since this stock analysis *job* is so time intensive, you would like it to be done in background mode - so, cook up a thread. Think of the thread as the *worker* you specify/create that is going to execute your *job*.

When the run() method is called, that's when your *job* starts, that is, the new call stack begins with the run() method.

The Java Thread class objects serve as an interface to the underlying

### Define and Instantiate a Thread

There are two ways to define and instantiate a thread

*Extend the java.lang.Thread* class and *@Override* the run() method. (why not just always extend the Thread class when you need a worker thread?)

```
class MyThread extends Thread {public void run() {System.out.println("Whahoo
job running in MyThread")}}
```

*Implement the Runnable interface*

```
class MyRunnable implements Runnable {public void run(){Sys-
tem.out.println("Yeah job running in MyRunnable")}}
```

So, either way, I now have some code that can be run by a thread of execution

### Instantiating a Thread

Recall an important comment earlier - Every TOE begins as an *instance of class Thread*. Whether your run() method is in a Thread subclass, or you have it in a Runnable implemented class, I still need a Thread *object* to do the work.

1. If you have extended the Thread class, then starting a thread is easy:

```
MyThread thread = new MyThread()
```

2. For a Runnable implemented class such as MyRunnable, I *still need a Thread instance* to run it.

```
class MyRunnable implements Runnable
{
. . .
public void run() { /* your job code goes here */}
}//end MyRunnable
```

This results in a division of labor: The Thread class handles the *thread-specific code* while your Runnable implemented class specifies the *job to be done ( that is, the code in the run() method)*.

Two Steps to instantiate a thread if you are using a Runnable class

1. Get a Runnable instance

```
MyRunnable runnable = new MyRunnable()
```

2. Get an instance of java.lang.Thread (recall, this is the *worker* component), and give that instance the job you want run.

```
Thread thread = new Thread(runnable); //tell the Thread object to use MY run()
method, as found in MyRunnable
```

So, to summarize: If I create a thread using the no-arg constructor, then the thread will call its own run() method. That is the way things work when you extend the java.lang.Thread class. For a Runnable though, I still need the Thread to do the work, but the work I want done, the job, is specified in the Runnable class.

## Thread States and Transitions

A thread can be in one of 5 states, New, Runnable, Running, Waiting/blocked, Dead

## Synchronizing Code:

Java uses locks to synchronize code, that is, to prevent corruption from competing threads accessing the same object. Every Java object has a built-in lock that is effective only when that object has synchronized *method* code. When you enter a synchronized non-static method, that thread automatically acquires the object lock. So, if thread 'A' has the lock, then no other thread can enter ANY synchronized code. That is, thread B can't enter any method that is synchronized. Releasing a lock means the thread holding that lock, exits the synchronized method. The lock is free at this point until some thread enters s synchronized method of that object.

Major Points:

1. Only methods (or blocks of code) can be synchronized, no variable or classes

2. Each object has a single lock
3. A class can have both synchronized and non-synchronized methods.
4. If thread A and thread B are about to execute a synchronized method in a class and both threads are using the same instance of the class to invoke the method, only one thread at a time can execute the method. So, if thread A gets the lock, thread B has to wait until A finishes its method call. So, we have that: Once a thread gets a lock on an object, no other thread can access ANY of the synchronized methods of that class.
5. If a class has both sync and non-synced methods, multiple threads can access the non-synced methods. (This is a good idea to protect only the bare minimum with synchronization)
6. A sleeping thread keeps its locks.
7. A thread can get multiple locks. For example, thread A enters a synced method thus acquiring lock A1. Immediately thread A invokes a synchronized method on another object and so getting another lock, A2. Once a thread has a lock on an object, it has a lock on all other synchronized methods of that object and so is not in competition with other threads.
8. You can sync on a block of code as opposed to a method.

### *A synchronized block example*

```
class SyncTest
{
public void foo()
{
System.out.println("This snippet is not synchronized");
synchronized(this) {System.out.println("This snippet IS synchronized"); }
}//end foo
}//end SyncTest
```

Note that I could have also placed the synchronized key word on the method (which would have synchronized all the component of the method)

### Static Methods Can be Synchronized

There is only one copy of the static data for a given class so only one lock is needed to protect it. The lock will be for the whole class however. The lock is associated with java.lang.Class which represents each Java class, and keeps a lock for that class. It is that object of the java.lang.Class that holds the lock, as shown below...

```
public static synchronized String bar() {return name;}
```

### Synchronization and Cooperation Between Threads

Threads cooperate by using the features of wait(), notify() and notifyAll(). The wait() method says for the thread to wait until some condition happens. When that condition occurs, I can use the notify() or the notifyAll() method to notify waiting threads to wake up and continue normal execution. notifyAll() wakes up all waiting threads while notify(), targets a particular thread from the waiting set.

These three methods, wait(), notify(), and notifyAll() must be called from a synchronized block or method (on the receiving object of these methods).

*Example: Cooperating Threads: Trying to withdraw more than what is in my account!*

In this example, with code shown below, *accountBalance* is the current balance in my account that is being accessed with a withdrawal request. In the *synchronized* method 'withdrawal()', you see a while loop that has a condition that is constantly tested. So long as the *accountBalance* is less than the *withdrawalRequest*, the condition is true and the loop continues to 'loop'. Every iteration encounters the *wait()* command and so the thread is blocked. Meanwhile, in some other part of the program, there are deposits (via method *public synchronized void deposit()*) going on. After every deposit, the deposit method *notifies waiting* threads that the accountBalance has changed. In particular, waiting threads attempting to process the withdrawal method are 'notified' to check the possibility that the *(accountBalance < withdrawalRequest)* might return false, and therefore code following the while loop is executed. That is, it is possible that now, (*acccountBalance >= withdrawalRequest)* and so there is now enough money in the account to make a legal withdrawal. Accordingly, the code after the while loop is now executed and the withdrawal proceeds.

```
public synchronized void withdraw(int withdrawalRequest)
{
try
{
while (accountBalance < withdrawalRequest)// the condition to test
wait(); //if true, suspend this accessing thread, since not enough money
}
catch(InterruptedException ex) {}
accountBalance -= withdrawalRequest; //go here if condition is false
}//end while
}//end withdraw()
```

**** meanwhile ***** In another part of the program or in another class, there are deposits that are happening constantly, on different threads. When the *accountBalance* is updated as in the method below, all *waiting* threads are notified so they can check their conditions to see if they can now continue.

```
public synchronized void deposit (int amount)
{
accountBalance += amount;
notifyAll(); // notify all waiting threads to check their conditions
bly resume processing
}//end deposit(
```

# Serialization

## Explore Serialization in Java

### Initial Ideas

Objects can be 'dehydrated' and then hydrated, they can be flattened and then inflated, they can be freeze-dried and then thawed. All by way of saying that objects can be transformed into formats suitable for saving and then retrieving/recreating. What's tough is when an object is saved at one period of time and then retrieved, but the retrieval program has changed in the meantime, rendering the original format in-correct. We will look at this 'versioning' issue later.

Objects have state and behavior which might be worth saving from session to session. The behavior is specified in the *class*, while the state lives within the individual *object instance*. If you want to save the state/behavior of object there are several ways to go:

1. Use *serialization* if your data will be only used by the program that serialized it. This mode writes a special file that holds the serialized (flattened) objects. Your program then reads this data from file back into memory and re-constitutes the object as a respectable member of the heap.
2. Us a plain text file if other programs will have to read in your data. In this case, write a file, using delimiters such as a comma (CSV files) or spaces or anything else your clientele knows how to parse!
3. Use other formats to save the object such as an XML file. We will look at XMLEncode, XMLDecode as a way to save off your data into XML formats.

### How to serialize an object to a file

1. Make a *FileOutputStream (the connector stream)* object and target the desired type of output file

```
FileOutputStream fos = new FileOutputStream("MyGame.ser"); // this will create
the MyGame.ser file
```

2. Make an *ObjectOutputStream*

```
ObjectOutputStream os = new ObjectOutputStream(fos); // chain the FileOutput-
Stream object
```

Note that *ObjectOutputStream* lets me write an object BUT it can't directly connect to a file. This is the *Decorator* pattern in action and involves chaining the *ObjectOutputStream* to the *FileOutputStream*.

3. Write the object (s)

```
os.writeObject(/* your object reference #1 here */);
os.writeObject(/* your object reference #2 here */);
```

4. Close the *ObjectOutputStream*

```
os.close(); // closing the top most stream, closes those that are enclosed by
it, so that the FileOutputStream is also closed.
```

### Stream Ideas

Java I/O, *java.io.**; has a very comprehensive set of classes that assist you in performing Input/Output functions on your data to transform it from *sources*, to *destinations*. Despite the number of I/O classes and their perceived complexity, the I/O package of classes is based on a couple of simple ideas - *connected data streams and chained data streams.* These 'streams' are abstractions that al-

low you to think at the level of simply specifying where the data can be found and where you would like it to go. You will see that you can first use *connection streams* to connect source to destination and then chain these to higher level *chain streams* to get more comprehensive method capability.

The first kind of stream: *connection streams,* represent a connection to a source or destination (like to/from a file or socket or even an in-memory location), while *chain streams* provide higher level methods but still depend on a connection stream being present.

For example, FileOutputStream is a low level *connection stream* I/O class that can only do one thing, write bytes to a file. If I want to write larger aggregates of data, such as a whole object, I can chain the FileOutputStream to another I/O stream, a *chain stream* with a call to ObjectOutput-Stream. ObjectOutputStream has more capability than the raw FileOutputStream so that when I call ObjectOutputStream on an object, that object is serialized (flattened) first and then chained to the FileOutputStream where its equivalent bytes are written to file.

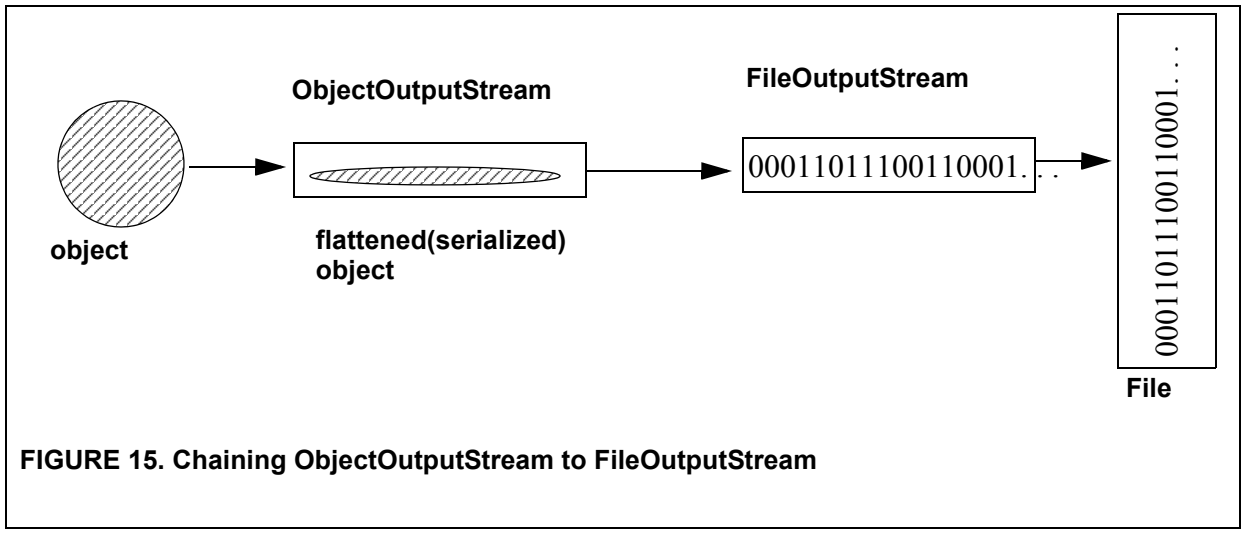



**FIGURE 15. Chaining ObjectOutputStream to FileOutputStream**

# JavaBeans and XML Encoding

This section discusses JavaBeans in the context of their usage in saving the state of JavaBean compatible classes. JavaBeans are Java classes that conform to a particular convention regarding constructors and accessors. A JavaBean is a Java object that is serializable, has a no arg constructor and allows access to properties via getters and setters. In order to qualify as a JavaBean the following must happen: (These constraints are mostly for the benefit of automated tools that can then more easily use, replace, and connect JavaBeans

1. The class must have a public default constructor (no argument).
2. The class properties must be accessible using get, set, is (used for Boolean properties instead of get) that follow a standard naming convention.
3. The class must be Serializable.

## Long Term Persistence - XMLEncoder, XMLDecoder

LOng term persistence is a model that lets beans (actually more general than beans can be saved) be saved in XML format. The XMLEncoder class is designed to write output files for textual representations of Serializable objects.

```
XMLEncoder encoder=
      new XMLEncoder(new FileOutputStream("beanArchive.xml"));
      encoder.writeObject(new MyObject());
      encoder.close();
```

The XMLDecoder class is designed to read back an XML document created by XMLEncoder.

```
XMLDecoder decoder =
      new XMLDecoder(new FileInutStream("beanArchive.xml"));
Object = decoder.readObject();
      decoder.close();
```

The XML bean archive has a special syntax that lets the XMLDecoder decode and recreate the xml-encoded object instance.

- an XML preamble (the standard <!xml . . . ?> header)
- a <java> tag to embody all object elements of the bean
- an <object> tag to represent a set of method calls needed to reconstruct and object from its encoding.
- tags to match primitive types <boolean><byte><char><short><int><long><float><double>
- a <class> tag to represent an instance of Class <class>java.swing.JFrame</class>
- an <array> tag to define an array <array class = "java.lang.String" length = "10"></array>

## Introspection

Introspection is the automatic process of analyzing a bean's design patterns to show the bean's properties, events, and methods.

The JavaBeans API supplies a set of classes and interfaces that provide introspection.

The BeanInfo interface in java.beans package defines a set of methods that allow bean implemen-

tors to provide explicit info about their beans such as descriptive name, hide methods, specify icons and in general configure their bean.

The getBeanInfo (beanName) class can be used by builder tools to provide detailed info about a bean. A call to getBeanInfo triggers the introspection process analyzing the beans classes and superclasses.

### Class Encoder

"An encoder is a class which can be used to create files or streams that encode the state of a collection of JavaBeans in terms of their public API's. The encoder, in conjunction with its persistence delegates, is responsible for breaking the object graph down into a series of Statements and Expressions which can be used to create it. "[Class Encoder spec]

A key method in this class is:

writeObject(Object o)

the spec says: Write the specified object to the output stream. The serialized form will denote a series of expressions, the combined effect of which will create an equivalent object when the input stream is read. By default, the object is assumed to be a JavaBean with a no-arg constructor whose state is defined by matching pairs of "setter" and "gettter" methods returned by the Introspector.

# Distributed Computing

## The OSI/TCP/IP Stacks

### OSI Stack

### TCP/IP Stack

# RMI

RMI is the Java answer to distributed system construction. So long as Java objects are at both ends of a connection, you can invoke methods on a remote service object just as you would a local object. As far as you, the client, know, that remote object appears to be local. Under the covers, RMI constructs a communication link based on socket I/O just as we have worked on in the recent past. Along with this, two surrogate objects are created to act as proxies for the remote service object. The proxy on the client is called the stub, while the proxy on the server host is called a skeleton.

### Key Components of th RMI Architecture

As a general background, I will call the object on the host server the *serviceIFImpl* object that has methods that a *client* object wishes to invoke.

- ServiceIF- this is a sub-interface of java.rmi.Remote that defines the methods of the service object that can be invoked by clients. This interface lays out what the client *could* invoke from the remote object.
- SeviceIFImpl - A *class* that implements the ServiceIF (remote object interface). Provides the body of the interface methods.
- Server object - A 'launcher class that instantiates a ServiceIFImple object, creates a registry, and binds the ServiceIFImple object to the registry. This bound object will become the 'service stub'.
- RMI registry - this utility registers remote objects and gives naming services for locating objects. The client will query this registry to discover and download the server stub.
- Client program - this is the program that wants to invoke methods on the remote server object.
- Service skeleton - provides the communication path between stub and the actual server object.


### How does the RMI System Work?

Ok, the plan is the following:

1. Define a *server object interface* that describes (abstractly) the methods that will be offered.

```
public interface ServiceInterface extends Remote
{ public void service1( . . .) { . . .}
//other methods that can be invoked
```

```
}//end interface ServiceInterface
```

2. Define a class that implements these methods - (an instance of this is a server object).

```
public class ServiceInterfaceImpl extends UnicastRemoteObject implements Ser-
viceInterface
{
public void service1(. . .) throws RemoteException
{ /* implement this method, that is, provide the body of code */}
// other method implementations
}//end class ServerInterfaceImpl
```

Note that the *UnicastRemoteObject* is required to integrate with TCP streams between two points on the net. Using UnicastRemoteObject automatically makes the object available for RMI access with JRMP(Java remote Method Protocol).

3. Now instantiate a service object from the ServiceInterfaceImpl class and register it with the *RMI registry.*

```
ServiceInterfaceImpl serverIF = new ServiceInterfaceImpl(. . .);
Naming.rebind( "rmi://host:port/name":,serverIF);
. . .
```

4. Create a client and direct it to locate the remote objects interface, download it and then use it to invoke the desired services on the remote server object.

```
Remote remoteObj = Naming.lookup("rmi://host:port/name")
ServiceInterfaceImple server = (ServiceInterfaceImpl)remoteObj;
serverIF.service1(. . . )
```

5. Security is not enabled if the RMI code is on the *local class path*. Otherwise, a policy file must be constructed and placed in your home directory. Below is an example of such a file. On a Windows box, I placed this file in my home directory at: *Documents and Settings\rrucker\java.polic*y

```
grant {
       permission java.net.SocketPermission "*:1024-65535", "connect,accept";
        permission java.net.SocketPermission "*:80", "connect";
        permission java.net.SocketPermission "*:RMIPortToUse", "con-
nect,resolve";
    };
```

# References

Arnold, K., J. Gosling, and D. Holmes (2000) *The Java Programming Language 3rd ed*. Addison Wesley.

Arnold, K., J. Gosling, and D. Holmes (2006) *The Java Programming Language 4th ed*. Addison Wesley.

Calvert, K. , Donahoo, M.(2008) TCP/IP Sockets in Java, Morgan Kaufmann.

Deitel, P. Deitel, H.(2012) Java How to Program (9th ed.) Prentice Hall

Liang,Y. 5th edition(2005), Pearson

Jendrock, E. et. al., (2011), *The Java EE 6 Tutorial* Fourth edition, Sun Publishing

Sierra, K. Bates, B. (2005) *Head First Java*, O'Reilly

Wirfs-Brock, Rebecca (200x) *Object Design*, **** TBD