tutorial

Saturday, December 21, 2019 11:35 PM

Chapter 1: An Introduction to Java

Buzzwords

Simple:
A cleaned-up version of C++ syntax.

Object-oriented
Focus on the data(=objects) and on the interfaces to that object.

Distributed
Has an extensive library of routines of coping with TCP/IP protocols, java applications can open and access objects across the Net via URLSs with the sane ease as when accessing a local file system .

Robust
Java is intended for writing programs that must be reliable in a variety of ways, java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. The java compiler detects many problems that in other languages would show up only at runtime.

Secure
Java enables the construction of virus -free, tamper-free systems. From the beginning, Java was designed to make certain kinds of attacks impossible.

- Overruning the runttime stack - a common attack of worms and viruses
- Corrupting memory ousitde its own process space
- Reading or writing files without permission

Untrusted code was executed in a sandbox environment where it cold not impact the host system.
Java code is incapable of escaping from the sandbox.
Java browser plug-ins no long trust remote code unless it is digitally signed and users have agreed to its execution.

Architecture-neutral
The compiler generates an architecture-neutral object file format - the compiled code is executable on many processors, given the presence of the Java runtime system. (generating bytecode instructions, are designed to be both easy to interpret on any machine and easily translated into native machin code on the fly=at runtime)
Java's virtual machine increases security cuz it can check the behavior of instruction sequences.

Portable
The libraries that are a part of the system define portable interfaces. The Java libraries do a great job of letting you work in aplatform-independent manner. (if the user interface toolkit has since been replaced for too many times, then portability across platforms remains an issue)

Interpreted
The Java interpreter canexecute Java bytecode directly on any machine to which the interpreter has been ported. Linking is a more incremental and lightweight process makes the development process can be much more rapid and exploratory.

High-Performance
While the performance of interpreted bytecodes is usually more than adquate, there are situations where higher performance is required. The bytecodes can be translated on the fly in to machin code for the particulatr CPU the application is running on. (just-in-time compiler, outperfrom)

Mutlthreaded
Benefits are better interactive responsiveness and real-time behavior. Concurrent programming was needed to make sure the user interface didn't freeze. Java make concurrency manageable even it's never easy.

Dynamic
It was designed to adapt to an evolving environment, libraries can cfreely add new methods and instance variables without any effect on their clients.

Chapter 2: The Java Programming Environment

JDK things

Environment

Chapter 3: Fundamental Programming Structures in Java

Data types

Integer types
Numbers without dractional parts, negative values are allowed
Four integer types: int(4 bytes), short(2 bytes), long(8 bytes), byte(1 byte)
Floating-Point types
Numbers with fractional parts
Two dloating-point types: float(4 bytes), double(8 bytes)
Char type
Some Unicode characters
unicode encoding scheme
\b(backspace), \t(tab), \n(linefeed), \r(carriage return), \"(double quote), \'(single quote), \\(backslash)
Boolean type
Used for evaluating logical conditions, you cannot convert between integers and boolean values.
Two values: false and true

Variables

Initializing variables:
`Type VariableName`
A variable name must begin with a letter and must be a sequence of letters or digits.
Use the keyword *final* to denote a constant, name in all uppercase.

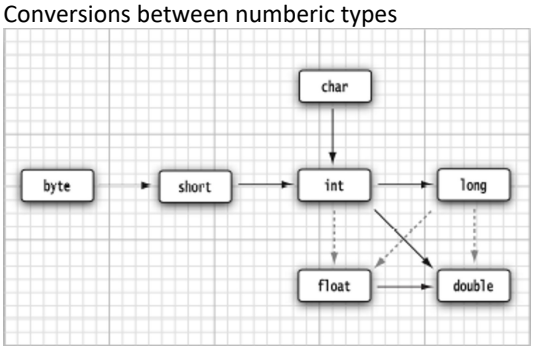
Keyword strcitfp:
public static `strictfp` void main(String[] args)
Then all instructions inside the main method will use strict floating-point computations.
If you tag a class as `strictfp`, then all of its methods must use strict floating-point computations.

Operators

Casts
Syntax: give the target type in parentheses, followed by the variable name.
Conversions in which loss of information is possible are done by means of *casts*. (type) variable

Import static java,lang.Math.*;
Then you can avoid the Math prefix;

Conversions between numberic types



Increment and decrement operators: += means i = i+1, i++, ++I;
Relational and boolean operators: &&, || , ==, !=
condition ? expr1(true): expr2(false)
Bitwise operators: & ("and") | ("or") ^ ("xor") ~ ("not")

Enumerated types(a variable holds a set of values):
enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE}
Size s = Size.MEDIUM;

Table 3.4 Operator Precedence

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

Strings

String e = "";
e.substring(a,b); /**get chars from a to b*/

Concatenation

Use + to join two strings
e.join("", "");

Strings are immutable

Use the equals method to test whether two strings are eual
.equals() or .equalsIgnoreCase();

Empty and null strings
str == null or str.length() == 0

To get the true length: int cpCount = e.codePointCount(0, e.length());
Return the code unit at a position n: char l = e.charAt(n);
To get at the ith code point:

```
Int index = e.offsetByCodePoints(0, i);
```

```
int cp = e.codePointAt(index);
```

Build strings:

```
StringBuilder builder = new StringBuilder();
Builder.append();
String completedString = builder.toString();
```

Input and Output

Reading input:

```
Scanner in = new Scanner(System.in);
String name = in.nextLine();
```

To read a single word, .next();
To read an integer, .nextInt(); or .nextDouble();
Cuz the Scanner class is defined in the java.util, you need to import by *import java.util.*;*
For passwords, use Consle class:

```
Console cons = System.console();
String username = cons.readLine("Username: ");
char[] passwd = cons.readPassword("Password: ");
```

Outputting format:
%m.nType
You can use the static String.format method to create a formatted string without printing it

File input and output:

```
Scanner in = new Scanner(Paths.get("read.txt"), "UTF-8");
PrintWriter out = new PrintWriter(Paths.get("output.txt"), "UTF-8");
```

Then access files by using System.in and System.out
(If you construct a Scanner with a file that does not exist or a PrintWriter with a file name that cannot be created, an exception occurs.)

Control Flow

Block scope:
A block or compound statement consists of a number of Java statements, surrounded by a pair of braces. Blocks define the scope of your variables. A block can be nested inside another block.
Conditional statements:
if (condition) {statements} else (condition) {statements}
Loops:
while (condition) {statements}, do {stataments} while (condition) {statements}
Determine loops:
for (;;) {statements}
Multiple selections:
switch (condition) {case 1: ; case 2: ; default: ;} break&continue

Big Numbers

Bit numbers: class in java.math packege: BigInteger and BigDecimal

Arrays

```
Arrays: int[] a = new int[]; int[] a = {}; new int[] {}; a = new int[] {};
```

Copy arrays: .copyOf();
Array sorting: .sort();
Multidimensional arrays: double[][] name; name = new double[][]; (you cannot use the array until you initialize it)

Chapter 4: Objects and Classes

Introduction to Object-Oriented Programming

Algorithms + data sttuctures = program
First they decided on the procedures for manipulating the data; then, they decided what structure to impose on the data to make the manipulations easier. OOP reverses the order: put the data dirst, then looks at the algorithms to operate on the data.

Classes
A class is the template or blueprint from which pbjects are made.

Objects
Three key characteristics of objects: behavior, state, identity. All objects that are instances of the same class share a family resemblance by supporting the same behavior. The behavior of an object is defined by the methods that you can call.

(the most common)relationships between classes

- Dependence("uses-a")
- Aggregation("has-a")
- Inheritance("is-a")

Using Predefined Classes

Import Class: LocalData, newYearsEve

Defining Your Own Classes

The simplest form: (every class has at least one constructor)

```
class ClassName{
     Filed1
     Filed2
     …
     Constructor1
     Constructor2
     …
     Method1
     Method2
     …
}
```

Encapusulation
Encapsulation (sometimes called information hiding) is a key concept in working with objects. Formally, encapsulation is simply combining data and behavior in one package and hiding the implementation details from the users of the object. The bits of data in an object are called its instance fields, and the procedures that operate on the data are called its methods. A specific object that is an instance of a class will have specific values of its instance fields. The set of those values is the current state of the object. Whenever you invoke a method on an oject, its state may change. The key to making encapsulation work is to have methods never directly access instance fields in a class other than their own. Programs should interact with object data only through the object's methods.

The final modifier is particularly useful for fields whose type is primitive or an immutable class. (A class is immutable if none of its methods ever mutate its objects. For example, the String class is immutable.)

Static Fields and Methods

There is only one static field per class. In constract, each object has its own copy of all instance fields.
Static methods are methods that do noe operate on objects. However, a static method can access a static field. Use static methods in two situations:

- When a method doesn't need to access the object state because all needed parameters are supplied as explicit parameters (example: Math.pow).
- When a method only needs to access static fields of the class (example: Employee.getNextId).

The main mehtod is a static method.
It is impossible for a method to change a primitive type parameter.

Here is a summary of what you can and cannot do with method parameters in Java:

- A method cannot modify a parameter of a primitive type (that is, numbers or boolean values).
- A method can change the state of an object parameter.
- A method cannot make an object parameter refer to a new object.

Object Construction

Constructor:
Define the initial state of your objects
As you can see, the name of the constructor is the same as the name of the class. This constructor runs when you construct objects of the class—giving the instance fields the initial state you want them to have. A constructor can only be called in conjunction with the new operator. You can't apply a constructor to an existing object to reset the instance fields.

- A constructor has the same name as the class.
- A class can have more than one constructor.
- A constructor can take zero, one, or more parameters.
- A constructor has no return value.
- A constructor is always called with the new operator.

Overloading occurs if several methods have the same name but different parameters. The compiler picks the correct method by matching the parameter types in the headers of the various methods with the types of the values used in the specific method call. A compile-time error occurs if the compiler cannot match the parameters, either becaused there is no match at all or because there is not one that is better than all others.(the process of finding a match is called *overloading resolution*)
You cannot have two methods with the same names and parameter types but different return types
If you don't set a explicityly in a constructor, it is automatically set to a default value: numbers to *0, boolean* values to *false*, and object references to *null*.
this refers to the implicit parameter of a method.

If the static fields of your class require complex initialization code, use a static initialization block. All static field initializers and static initialization blocks are executed in the order in which they occur in the class declaration.

You have already seen two ways to initialize a data field:
- By setting a value in a constructor
- By assigning a value in the declaration

There is a third mechanism in Java, called an initialization block.

it is important that the resource be reclaimed and recycled when it is no longer needed. You can add a *finalize* method to any class. The finalize method will be called before the garbage collector sweeps away the object.

NOTE: The method call `System.runFinalizersOnExit(true)` guarantees that finalizer methods are called before Java shuts down. However, this method is inherently unsafe and has been deprecated. An alternative is to add "shutdown hooks" with the method `Runtime.addShutdownHook`—see the API documentation for details.

Packages

Package: a collection of classes
The main reason for using packages is to guarantee the uniqueness of class names.
Subpackages: You then use subpackages for different projects. For example, consider the domain **horstmann.com**. When written in reverse order, it turns into the package **com.horstmann**. That package can then be further subdivided into subpackages such as **com.horstmann.corejava**.

Class importation:
A class can use all classes from its own package and all public classes from other packages.
Once you use import, you no longer have to give the classes their full names.
Import by specific class make it more readable and clear for readers

Static imports:
A form of the **import** statement permits the importing of static methods and fields, not just classes.
To place classes inside a package, you must put the name of the package at the top of your source file, before the code that defines the classes in the package. If you don't put a package statement in the source file, then the classes in that source file belong to the default package. The default package has no package name.

The Class Path

Java compiler:
```
javac [ options ] [ sourcefiles ] [ classes ] [ @argfiles ]
```

来自 <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html>

you can use another mechanism, **package sealing**, to address the issue of promiscuous package access. If you seal a package, no further classes can be added to it.
Class files can also be stored in a JAR (Java archive) file. A JAR file contains multiple class files and subdirectories in a compressed format, saving space and improving performance. When you use a third-party library in your programs, you will usually be given one or more JAR files to include.

TIP: JAR files use the ZIP format to organize files and subdirectories. You can use any ZIP utility to peek inside `rt.jar` and other JAR files.

CAUTION: The `javac` compiler always looks for files in the current directory, but the `java` virtual machine launcher only looks into the current directory if the "." directory is on the class path. If you have no class path set, this is not a problem—the default class path consists of the "." directory. But if you have set the class path and forgot to include the "." directory, your programs will compile without error, but they won't run.

Variables must explicitly be marked private, or they will default to being package visible. This, of course, breaks encapsulation.

TIP: JAR files use the ZIP format to organize files and subdirectories. You can use any ZIP utility to peek inside `rt.jar` and other JAR files.

Documentation Comments

The class comment must be placed after any import statements, directly before the class definition.
Each method comment must immediately precede the method that it describes.
Method commends:

@param *variable* description
@return description
@throws *class* description
General comments:
@author name, name, name…
@version *text*
(next are good for all documents comments)
@since *text* : The text can be any description of the version that introduced this feature. For example, @since version 1.7.1
@deprecated *text* : This tag adds a comment that the class, method, or variable should no longer be used. The text should suggest a replacement. For example: @deprecated Use <code> setVisible(true)</code> instead
@see *reference* : This tag adds a hyperlink in the "see also" section. It can be used with both classes and methods; specify a hyperlink; supply the name of a class, method or variable; followed by a " character, then the text is displayed in the "see also" section. You can add multiple @see tags for one feature, but you must keep them all together.

Package comments:
to generate package comments, you need to add a separate file in each package directory. You have two choices:

1. Supply an HTML file named package.html. All text between the tags <body>...</body> is extracted.
2. Supply a Java file named package-info.java. The file must contain an initial Javadoc comment, delimited with /** and */, followed by a package statement. It should contain no further code or comments.

Overview comments:
You can also supply an overview comment for all source files. Place it in a file called overview.html, located in the parent directory that contains all the source files. All text between the tags <body>...</body> is extracted. This comment is displayed when the user selects "Overview" from the navigation bar.

Class Design Hints

1. Always keep data private
2. Always initialize data
3. Don't use too many basic types in a class:
   The idea is to replace multiple related uses of basic types with other classes. This keeps your classes easier to understand and to change. Replace a type with a class.
4. Not all field needs inidividual field accessors and mutators
5. Break up classes that have too many responsibilities
6. Make the names of your classes and methods reflect their responsibilities.
7. Prefer immutable classes

Chapter 5: Inheritance

Classes, Superclasses and Subclasses

Use the Java keyword **extends** to denote inheritance.
The keyword **extends** indicates that you are making a new class that derives from an existing class. The existing class is called the **superclass, base class, or parent class**. The new class is called the **subclass, derived class, or child class**.
When designing classes, you place the most general methods in the superclass and more specialized methods in its subclasses. Factoring out common functionality by moving it to a superclass is common in object-oriented programming.
If the Manager methods want to access those private fields, they have to do what every other method does—use the public interface, in this case the public getSalary method of the Employee

class.
We need to indicate that we want to call the getSalary method of the Employee superclass, not the current class. You use the special keyword super for this purpose.

```
public double getSalary()
{
     double baseSalary = super.getSalary();
     return baseSalary + bonus;
}
```

A subclass can add fields, and it can add methods or override the methods of the superclass.
However, inheritance can never take away any fields or methods.
The constructor is invoked with the special **super** syntax.

The collection of all classes extending a common superclass is called an *inheritance hierarchy.*
The path from a particular class to its ancestors in the inheritance hierarchy is its *inheritance chain*.

"is-a" rule:
States that every object of the subclass is an object of the superclass
Subsitution priciple, that principle states that you can use a subclass object whenever the program expects a superclass object
For example, you can assign a subclass object to a superclass variable. That means you can create a subclass object by using a superclass class.
In the Java programming language, object variables are polymorphic. A variable of type Employee can refer to an object of type Employee or to an object of any subclass of the Employee class (such as Manager, Executive, Secretary, and so on).

It is important to understand exactly how a method call is applied to an object.

1. The compiler looks at the declear type of the object and the method name. (there may be multiple methods, all with the same name, but with different parameter types) The compiler enumerates all methods called ** in the class and all accessible methods ** in the superclasses of the class (except private ones)
2. Next, the compiler determines the types of the arguments that are supplied in the method call. If among all the methods called ** there is a unique method whose parameter types are a best match for the supplied arguments, that method is chosen to be called. The process is called *overloading resolution*. If the compiler cannot find any method with matching parameter types or if multiple methods all match after applying conversions,, the compiler reports a error.
3. If the method is **private, static, final**, or a constructor, then the compiler knows exactly which method to call. This is called *static binding*. Otherwise, the method to be called depends on the actual type of the implicit parameter, and dynamic binding must be used at runtime.
4. When the program runs and uses dynamic binding to call a method, the virtual machine must call the version of the method that is appropriate for the actual type of the object refers.

When a method is actually called, the virtual machine simply makes a table lookup. (a method table for each class)

CAUTION: When you override a method, the subclass method must be *at least as visible* as the superclass method. In particular, if the superclass method is `public`, the subclass method must also be declared `public`. It is a common error to accidentally omit the `public` specifier for the subclass method. The compiler then complains that you try to supply a more restrictive access privilege.

Classes that cannot be extended are called *final* classes, and you use the **final** modifier in the definition of the class to indicate this. You can also make a specific method in a class **final**, then no subclass can override that method.

NOTE: Recall that fields can also be declared as `final`. A final field cannot be changed after the object has been constructed. However, if a class is declared `final`, only the methods, not the fields, are automatically `final`.

There is only one good reason to make a method or class final: to make sure its semantics cannot be changed in a subclass. We agree that it is a good idea to think carefully about final methods and classes when you design a class hierarchy.

To actually make a cast of an object reference, use a syntax similar to what you use for casting a numeric expression. Surround the target class name with parentheses and place it before the object reference you want to cast. For example: Manager boss = (Manager) staff[0];

There is only one reason why you would want to make a cast—to use an object in its full capacity after its actual type has been temporarily forgotten.

The compiler checks that you do not promise too much when you store a value in a variable. If you assign a subclass reference to a superclass variable, you are promising less, and the compiler will simply let you do it. If you assign a superclass reference to a subclass variable, you are promising more. Then you must use a cast so that your promise can be checked at runtime.
It is good programming practice to find out whther a cast will succeed before attempting it. Simply use the **instanceof**. Finally, the compiler will not let you make a cast if there is no chance for the cast to succeed.
To sum up:

- You can cast only within an inheritance hierachy
- Use **instanceof** to check before casting from a superclass to subclass

The dynamic binding that makes polymorphism work locates the correct method automatically. The only reason to make the cast is to use a method that is unique to managers, such as setBonus. In general, it is best to minimize the use of casts and the **instanceof** operator.

If you use the **abstract** keyword, you do not need to implement the method at all. For added clarity, a class with one or more abstract methods must itself be declared abstract. In addition to abstract methods, abstract classes can have fields and concrete methods.

TIP: Some programmers don't realize that abstract classes can have concrete methods. You should always move common fields and methods (whether abstract or not) to the superclass (whether abstract or not).

Abstract methods act as placeholders for methods that are implemented in the subclasses. When you extend an abstract class, you have two choices. You can leave some or all of the abstract methods undefined; then you must tag the subclass as abstract as well. Or you can define all methods, and the subclass is no longer abstract.
A class can even be decleared as **abstract** though it has no abstract methods.
Abstract classes cannot be instantiated. That is, if a class is declared as abstract, no objects of that class can be created. However, you can create objects of concrete subclasses.

Fields in a class are best tagged as **private**, and methods are usually tagged as **public**. Any features declared **private** won't be visible to other classes. A subclass cannot access the private fields of its superclass. When you want to restrict a method to subclasses only or, less commonly, to allow subclass methods to access a suerclass field. In that case, you declare a class feature as **protected**. A class may declare a method as **protected** if it is tricky to use. This indicates that the subclasses can be trusted to use the method correctly, but other classes cannot.
A summary of the four access modifiers in Java that control visibility:

1. Visible to the class only(private)
2. Visible to the world(public)
3. Visible to the package and all subclasses(protected)
4. Visible to the package -- the (unfortunate) default. No modifiers are needed

Object: The cosmic Superclass

The object class is the ultimate ancestor -- every class in Java extends Object
In Java, only the values of *primitive types* (numbers, characters, and boolean values) are not objects.
The **equals** method in the **Object** class tests whether one object is considered equal to another. The **equals** method, as implemented in the **Object** class, determines whether two object references are identical. This is a pretty reasonable default—if two objects are identical, they should certainly be equal.
Equal method has properties:

- **Reflexive**: for any non-full reference x, x.equals(x) should return true
- **Symmetric**: for any references should return true and vice versa
- **Transitive**: if x=y,y=z, then x=z
- **Consistent**: if value no change, then result don't change

For any non-null reference x, x.equals(null) should return false

Here is a recipe for writing the perfect `equals` method:

1. Name the explicit parameter `otherObject`—later, you will need to cast it to another variable that you should call `other`.
2. Test whether `this` happens to be identical to `otherObject`:

   ```
   if (this == otherObject) return true;
   ```

   This statement is just an optimization. In practice, this is a common case. It is much cheaper to check for identity than to compare the fields.
3. Test whether `otherObject` is `null` and return `false` if it is. This test is required.

   ```
   if (otherObject == null) return false;
   ```
4. Compare the classes of `this` and `otherObject`. If the semantics of `equals` can change in subclasses, use the `getClass` test:

   ```
   if (getClass() != otherObject.getClass()) return false;
   ```

   If the same semantics holds for *all* subclasses, you can use an `instanceof` test:

   ```
   if (!(otherObject instanceof ClassName)) return false;
   ```
5. Cast `otherObject` to a variable of your class type:

   ```
   ClassName other = (ClassName) otherObject
   ```
6. Now compare the fields, as required by your notion of equality. Use `==` for primitive type fields, `Objects.equals` for object fields. Return `true` if all fields match, `false` otherwise.

   ```
   return field1 == other.field1
      && Objects.equals(field2, other.field2)
      && . . .;
   ```

   If you redefine `equals` in a subclass, include a call to `super.equals(other)`.

TIP: If you have fields of array type, you can use the static `Arrays.equals` method to check that the corresponding array elements are equal.

An error is reported beacause this method doesn't @Override any method from the Object superclass

A hash code is an integer that is derived from an object. Hash codes should be scrambled—if x and y are two distinct objects, there should be a high probability that x.hashCode() and y.hashCode() are different. The hashCode method should return an integer (which can be negative).

toString method returns a string representing the value of this object

Generic Array List

ArrayList is a generic class with a type parameter. To specify the type of the element objects that the array list holds, you append a class name enclosed in angle brackets, such as ArrayList<Employee>.

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

Compability between typed and raw array list
Using cast may not work
You can tag the variable that receives the cast with the @SuppressWarnings("unchecked") annotation, like this:

```
@SuppressWarnings("unchecked") ArrayList<Employee> result =
      (ArrayList<Emloyee>) employee.DB.find(query);
```

Object Wrappers and Autoboxing

All primitive types have class counterparts (eg. A class **Intger** corresponds to the primityive type int), these kinds of classes are usually called *wrappers*.

The wrapper classes are immutable -- you cannot change a wrapped value after the wrapper has been constructed. They are also final, so you cannot subclass them.
If the type paramter inside the angle brackets cannot be a primitive tyoe, here, the wrapper class comes in, it is okay to declar an array list of this wrapper class objects.

CAUTION: An `ArrayList<Integer>` is far less efficient than an `int[]` array because each value is separately wrapped inside an object. You would only want to use this construct for small collections when programmer convenience is more important than efficiency.

There is a useful feature that makes it easy to add an element of tyoe in to an ArrayList<Integer>.
The call: list.add(3); is automatically translated to list.add(Integer.valueOf(3)); This conversation is called autoboxing. Conversely, when you assign an Integer object to an int value, it is automatically unboxed. That is compiler translates int n = list.get(i); into int n = list.get(i).intValue(); Automatic boxing and unboxing even works with arithmetic expressions. The compiler automatically do sth then unbox the object, get the result then box it back.
Operators applied to wrapper objects only tests about the objects' memory locations. (use equals method plz)
If you mix a wrapper object and a primitive type in a conditional expression, then the wrapper object value is unboxed, promoted to another type and boxed into a type thing. (like Integer boxed into a Double)
The compiler insters the necessary calls when it generates the bytecodes of a class. The virtual machine simply executes those bytecodes.

Convert a string to ints: int x = Integer.parseInt(s); // a static method,

Ive said the wrapper objects are immutable, then don't get with shit about changing the value.

Methods with a Variable Number of Parameters

Methods with a variable number of parameters are called "varargs" method.

Enumeration Classes

All enumerated types are subclasses of the class Enum. They inherit a number of methods from that class. The most useful one is toString, which returns the name of the enumerated constant.
Each enumerated type has a static values method that returns an array of all values of the enumeration.

Reflection

The reflection library gives you a very rich and elaborate toolset to write programs that manipulate Java code dynamically. This feature is heavily used in JavaBeans, the component architecture for Java (see Volume II for more on JavaBeans). Using reflection, Java can support tools like those to which users of Visual Basic have grown accustomed. In particular, when new classes are added at design time or runtime, rapid application development tools can dynamically inquire about the capabilities of these classes.
A program that can analyze the capabilities of classes is called reflective. The reflection mechanism is extremely powerful. You can use it to:

- Analyze the capabilities of classes at runtime
- Inspect objects at runtime -- for example, to write a single toString method that works for all classes
- Implement generic array manipulation code
- Take advantage of method objects that work just like fucntion pointers in lanaguage such as C++

The getClass() method in the Object class returns an instance of Class type

A primer on catching exceptions
When an error occurs at runtime, a program can "throw an exception." throwing an exception is mor

flexible than terminating the program because you can provide a **handler** that "catches" the exception and deals with it. If you don’t provide a handler, the program still terminates and prints a message to the console.
There are two kinds of exceptions: **uchecked** exceptions and **checked** exceptions.
Cheched exceptions: the compiler checks that you provide a handler
Unchechecked exceptions: the compiler does not check whether you provide a handler for these errors.
Sytax:

```
Try{
     Statements that might throw exceptions
} catch(exception e) {
     Handler action
}
```

Design Hints for Inheritance

Useful when using inheritance

1. Place common operations and fields in the superclass
2. Don’t use protected fields
3. Use inheritance to model the "is-a" relationship
4. Don’t use inheritance unless all inherited methods make sense
5. Don’t change the expected behavior when you override a method
6. Use polymorphism, not type information
7. Don’t overuse reflection