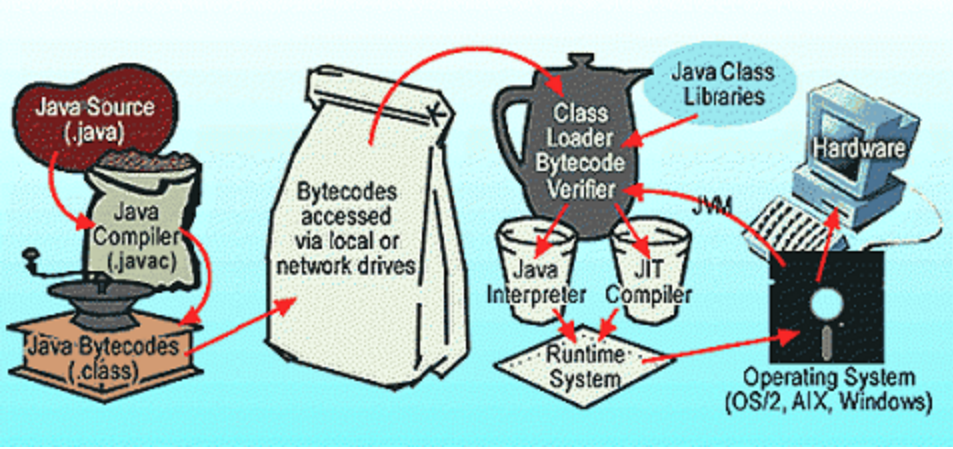
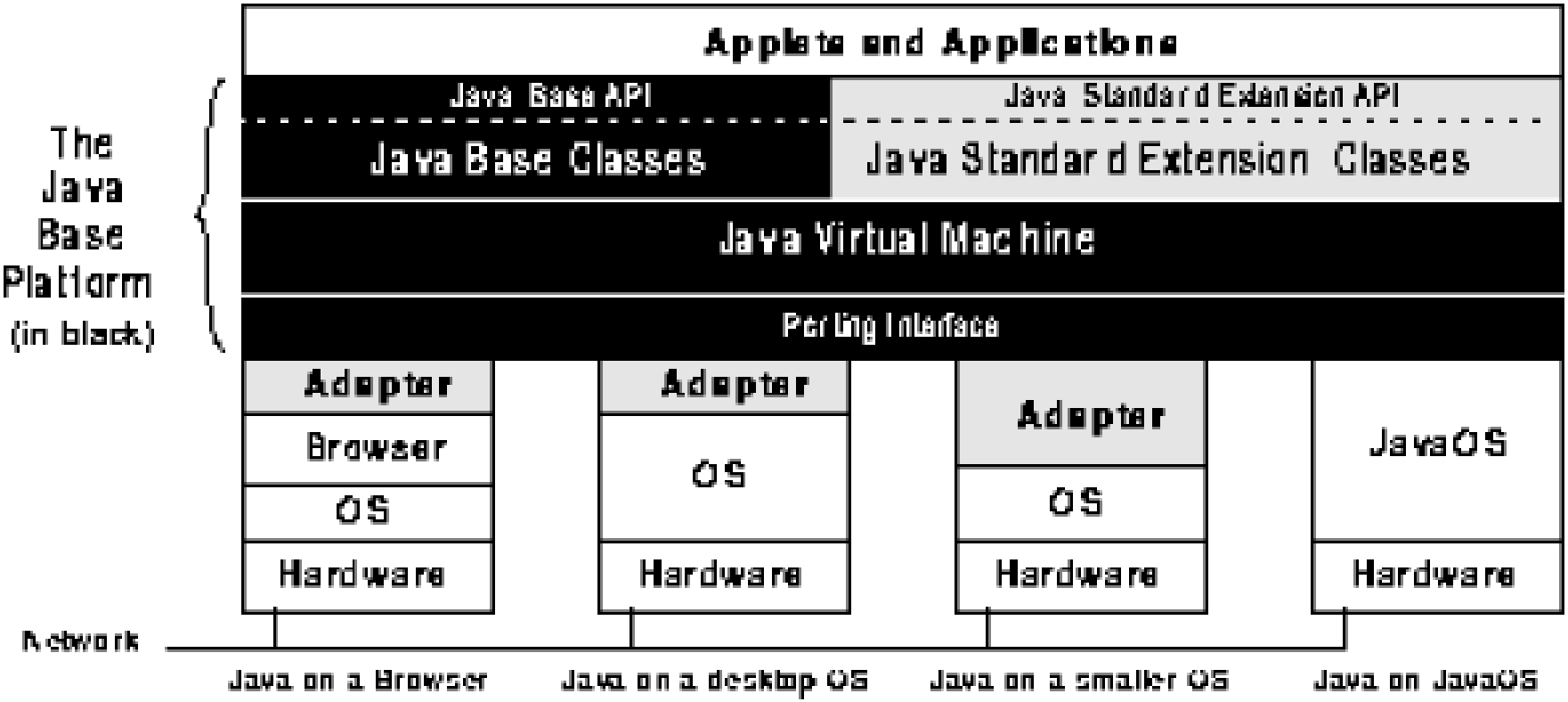
The Java Programming Environment



Java Platform

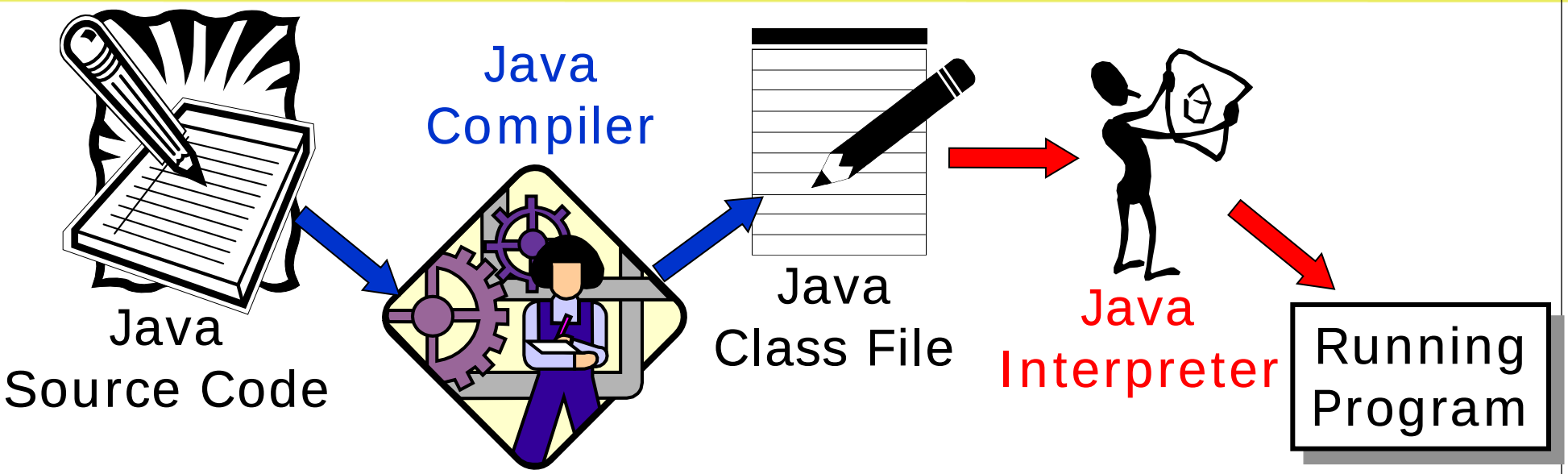


- There are three main platforms for Java:
  - **Java SE** (short for Standard Edition) – runs on desktops and laptops
  - **Java ME** (short for Micro Edition) – runs on mobile devices such as cell phones
  - **Java EE** (short for Enterprise Edition) – runs on servers

Java Terminology

- Java Virtual Machine (JVM) – An abstract machine architecture specified by the Java Virtual Machine Specification, Second Edition by Tim Lindholm and Frank Yellin (ISBN: 0201432943)
- Java Runtime Environment (JRE) – A runtime environment which implements Java Virtual Machine, and provides all class libraries and other facilities (such as JNI) necessary to execute Java programs. This is the software on your computer that actually runs Java programs.
  - Note: These two terms (JVM and JRE) are often used interchangeably, but it should be noted that they are not technically the same thing
- Java Development Kit (JDK) – The basic tools necessary to compile, document, and package Java programs (javac, javadoc, and jar, respectively). *The JDK includes a complete JRE*

Java Development Cycle



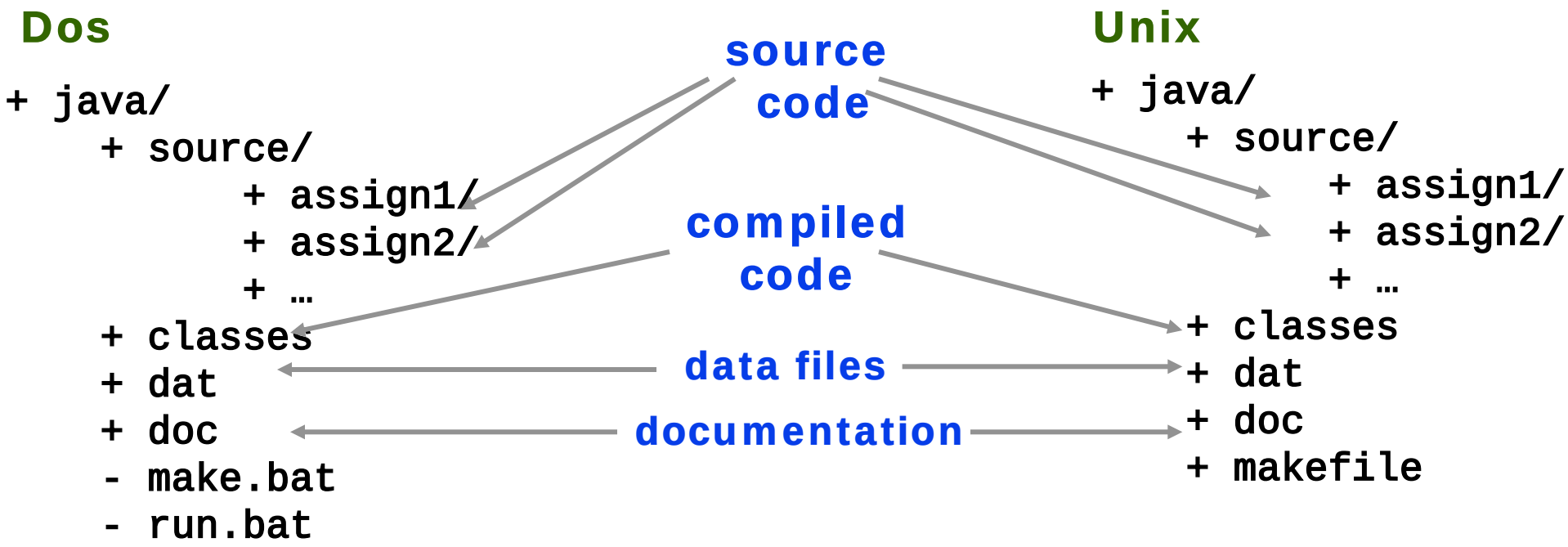
- The Java compiler translates Java source code into a special representation called bytecode
- Java bytecode can be thought as the machine code for a fictional machine called the Java Virtual Machine
- The Java interpreter translates the bytecode into machine language code and executes it
- The use of bytecode makes Java platform independent

Setting up your Directories for Java Development

- First, do you know what the difference is?
  - Source file?
  - Class file?
  - "jar" files?
  - Compiler?
  - "javac"?
  - "java"?

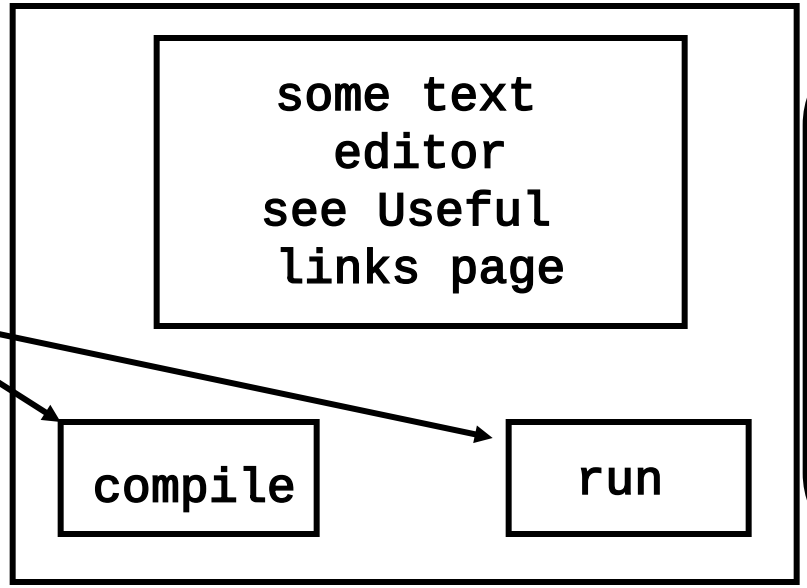
- 2-Ways to do assignments:
  - Option 1: Log into your Unix account from the lab or by calling in (Advantages: automatic backups, Know it works right, little or no set up time Disadvantages: Slow, new and scary)
  - Option 2: Work on a Windows PC and *ftp* your files when you are done to your Unix account. (Advantages: Usually a GUI based interface, friendly Disadvantages: Have to zip, ftp and unzip, and test in the Unix environment)

How to Organize your Folders (Windows or Unix)



Setting up the Windows - Unix

Dos windows



Without having two DOS windows you can type "*doskey*" at the start of the first DOS window and use the arrow keys to cycle commands

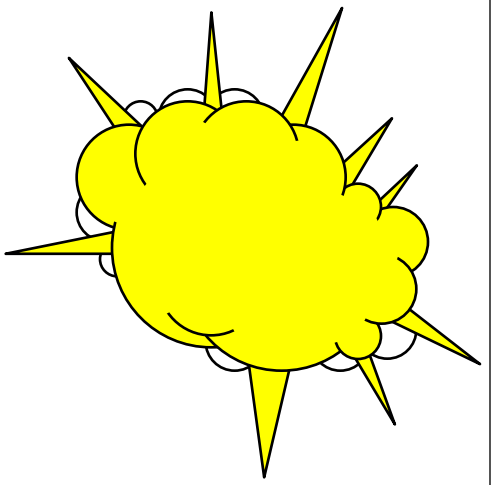
Setting up windows in Unix

Duplicate sessions if you are "telneted" and working from home arrow keys are set up with history

Setting up your Java Environment

- After downloading and installing the most recent JDK, you must set your CLASSPATH and JAVA_HOME environment variables. JAVA_HOME should be the location of your JDK installation

SUN or PC-Solaris Workstation



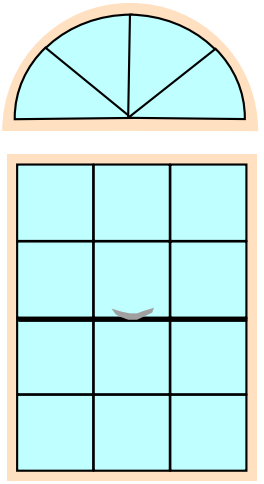
- Add the following lines the your .login file:

```
setenv PATH /usr/java/bin
```

- and the following lines in your .cshrc file:

```
unsetenv LC_CTYPE
unsetenv LANG
```

DOS/WINDOWS



- Java installer sets PATH and CLASSPATH automatically
- Troubleshooting tip: Put the following lines

```
SET PATH=%PATH%;C:\j2sdk1.5\bin;C:\j2sdk1.5\jre\bin
```

in your autoexec.bat

The Java Class Path

- The class path is a list of locations (folders) that are searched for classes when the Java Virtual Machine attempts to locate a referenced class
  - The Java compiler (the `javac` command) automatically looks in the current directory when compiling files, but the Java interpreter (the `java` command) only looks in the current directory if the class path is undefined: You would then be able to compile classes, but be unable to run them
- The `CLASSPATH` variable is an environment variable that specifies the class path on a given system
  - The `CLASSPATH` must be set and must include the current directory "." in order to run Java programs from the command line with the `java` command, otherwise you will get an error similar to:

```
Exception in thread "main"
java.lang.NoClassDefFoundError: SomeClass/java
```

  - The CLASSPATH variables simply contains a list of absolute directories, separated by a system dependent separator. On Windows a semicolon ";" is used to separate directories, on Linux, Unix, Mac OS X, and Solaris, a colon ":" is used to separate directories

Testing the Installation

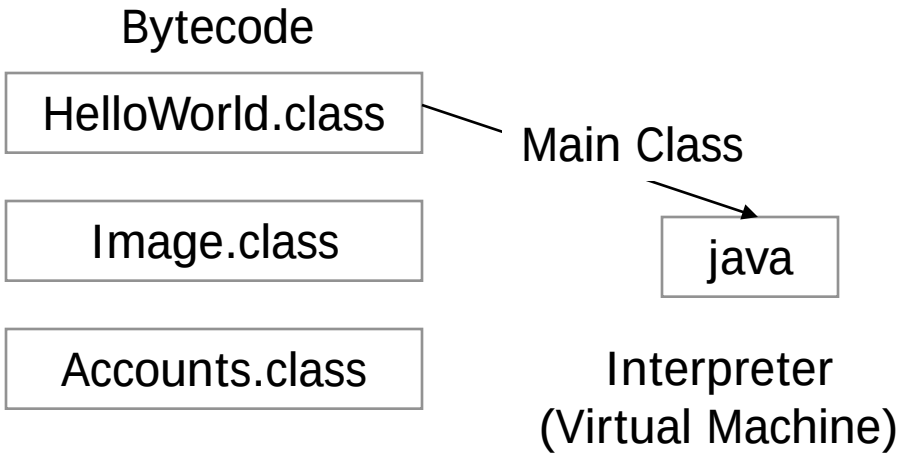
- Once you have installed the JDK, set the CLASSPATH and JAVA_HOME environment variables, and updated your PATH variable, type the following on a command line: **`javac -version`**

  You should get the following output: **`javac 1.6.0`**

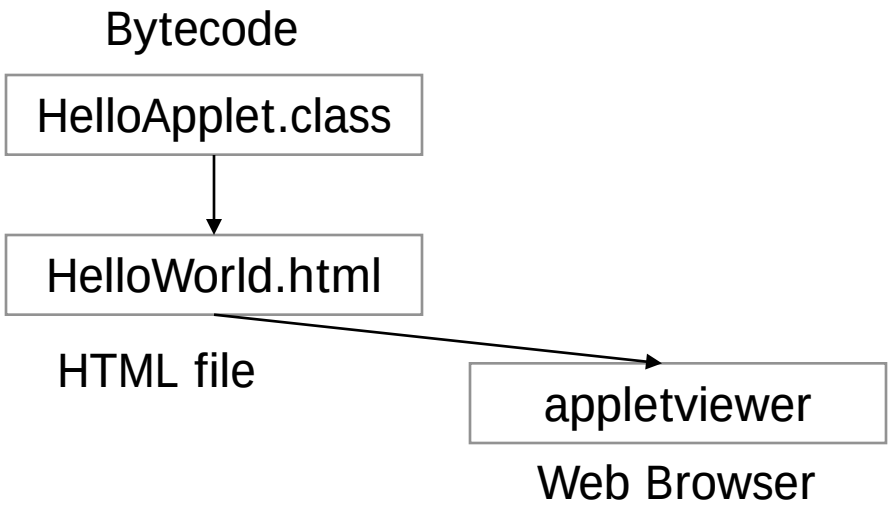
- JDK Important Locations
  - **`jdk/bin`** – contains command line tools such as `javac, java, javadoc,` and `jar`
  - **`jdk/jre`** – the location of the bundled JRE
  - The standard Java SE class libraries are located in `jre/lib/rt.jar`
  - Any class libraries (.jar files) located in the `jre/lib/ext` directory are also available to Java programs
    - You don't need to worry about adding **`jre/lib/rt.jar`** or **`jre/lib/ext`** to your classpath because both of these are searched automatically by both java and javac

Java Applications and Applets

- There are two types of Java applications
  - Java Applications
    - Run stand alone



  - Java Applets
    - Run within a web browser
    - Embedded within an HTML file
    - Have greater security restrictions



The "Hello World" Example

- Create a file named `HelloWorld.java`

```
public class HelloWorld
{
   public static void
    main(String[ ] args)
   {
     System.out.println("
      Hello World !!!");
   }
}
```

- Use
  - Notepad, Wordpad, DOS Edit
  - (g)vim, emacs, pico, nano
  - Eclipse, Jbuilder, Visual Age, Visual J++, CodeWarrior
- Source file rules
  - Each file can only contain one public class (e.g., `HelloWorld`)
  - The name of the file must be the same name of the class within it plus `.java` extension (e.g., `HelloWorld.java`)

Compiling a Java Program

- Compiling source files with `javac`
  - type to compile filename.java: `javac filename.java`
  - produces the class file: `filename.class`
- Executing the compiler in a command line environment:

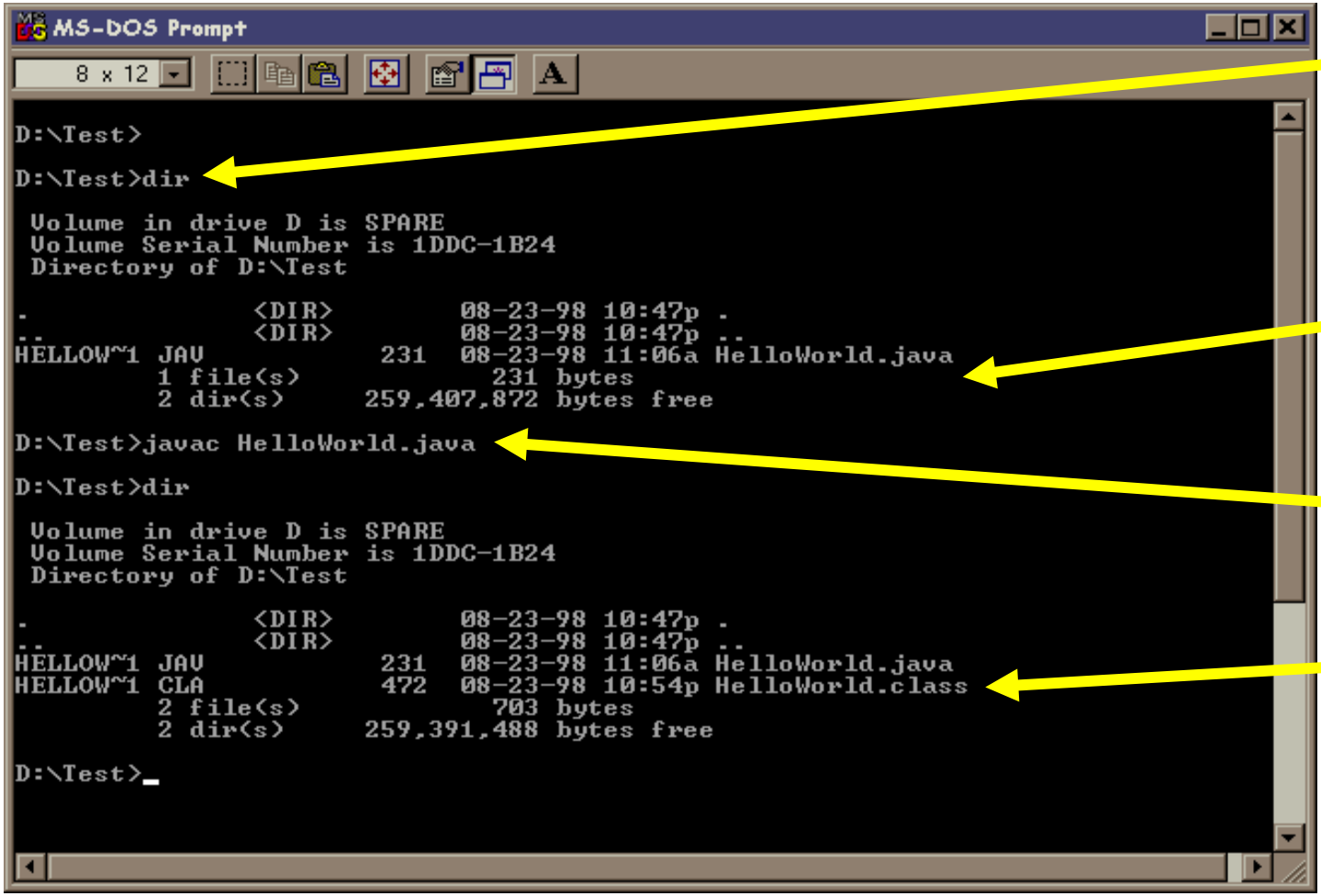
```
javac HelloWorld.java
```

- This creates a file called `HelloWorld.class`, which is submitted to the interpreter to be executed:

```
java HelloWorld
```

- The `.java` extension is used at compile time, but the `.class` extension is not used with the interpreter
  - A class file is produced for each class compiled
  - Other environments do this processing in a different way

How do I Compile the Java Code?



MS-DOS command to see directory of files

“.java” source file must be *.java

Running the compiler

Generated “.class” file

Running a Java Program

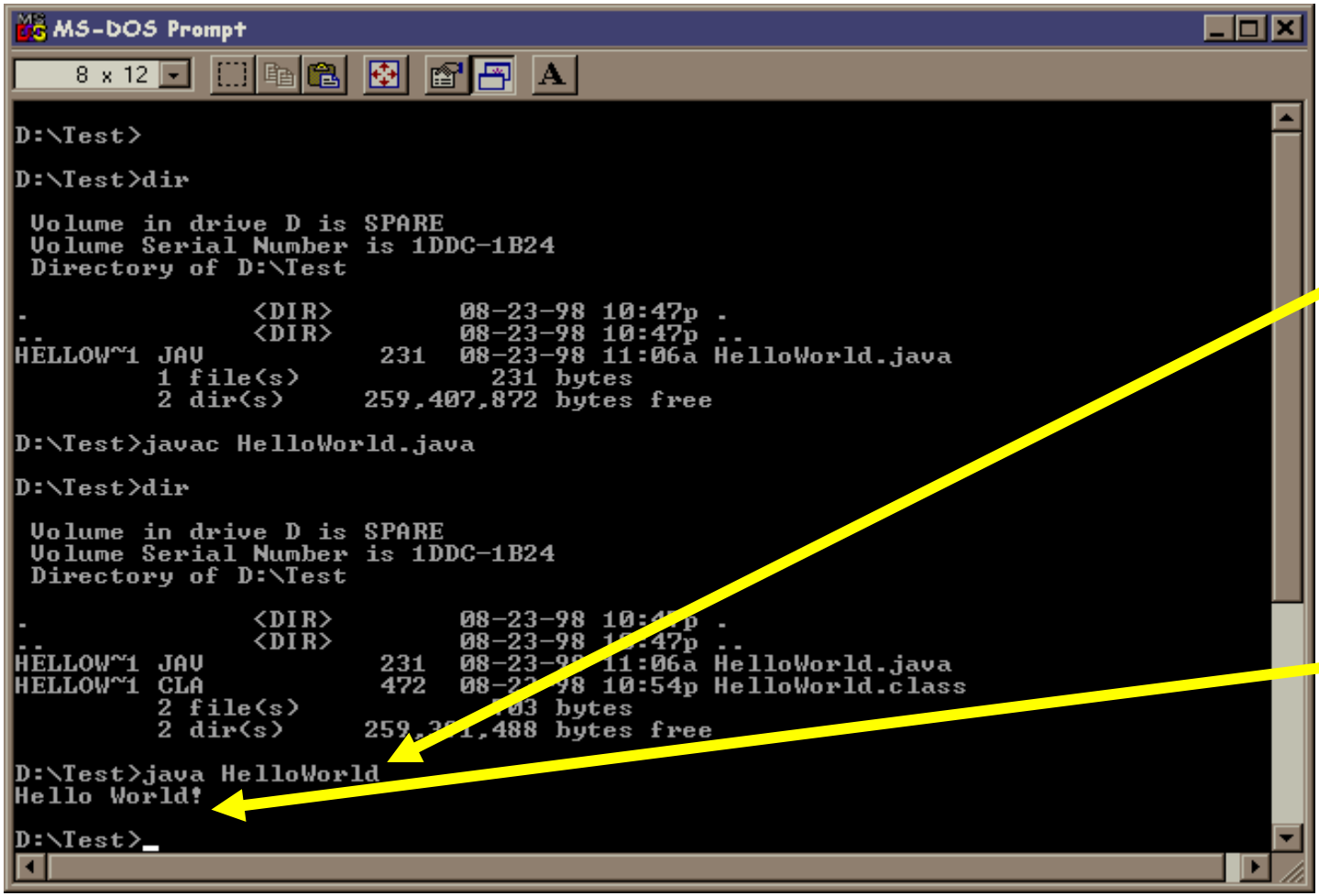
- Running class files by submitting them to the `java` interpreter
  - type to execute classname.class

    `java classname`
  - Note: only type the class name, **`.java`** or **`.class`** extensions are not needed
  - to execute **`HelloWorld.class, type:`**

    `java HelloWorld`
- The class passed to `java` must contain a **`main(String[] args)`** function
  - The `main` method accepts arguments on the command line when a program is executed

    `java Name_Tag John`
  - In Java, command line arguments are always read as a list of character strings
    - Each extra value is called command line argument

How do I run a Java Code?

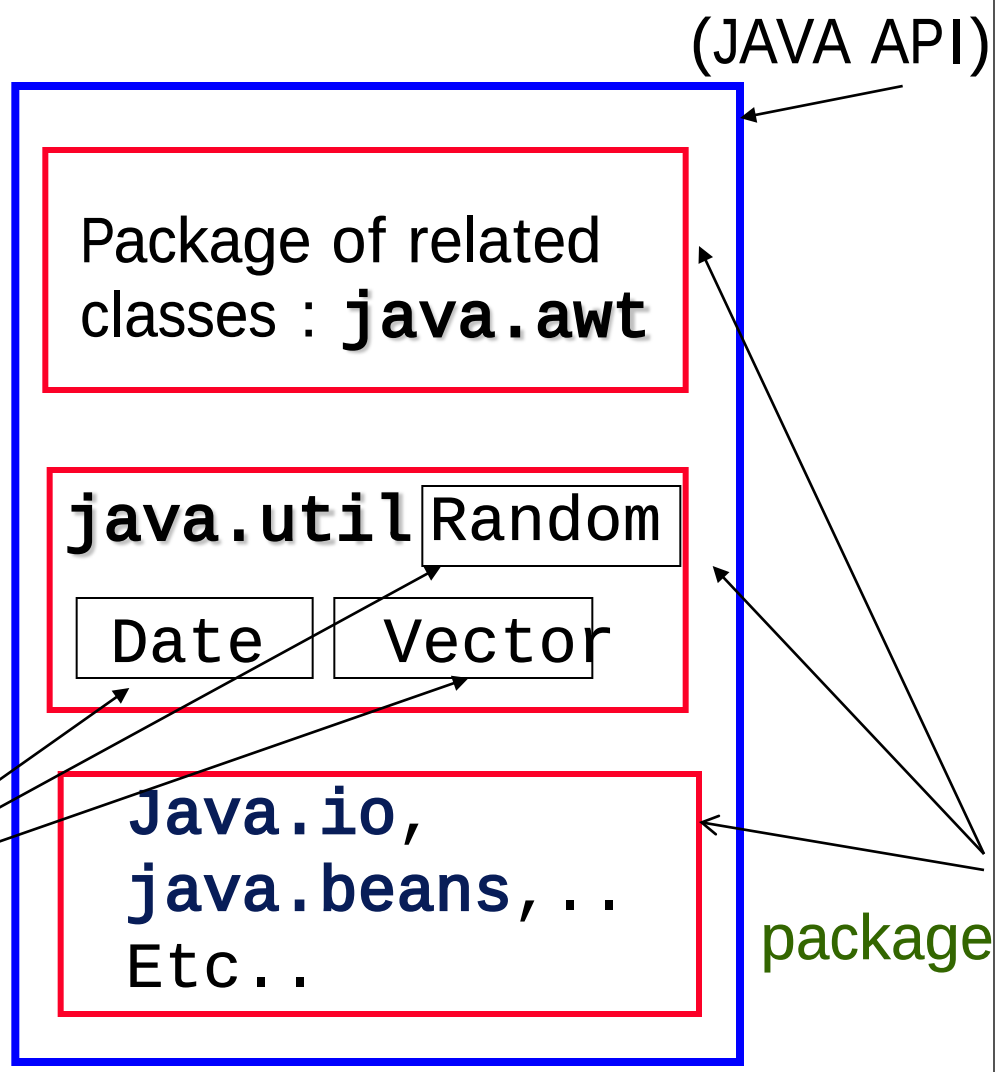


Running the Java interpreter

Program output, and finish

What are the Java Class Libraries and Packages?

- The Java API is a class library, a group of classes that support program development
- A `package` is used to group similar and interdependent classes together
  - The classes in the Java API are separated into packages
  - Each package contains a set of classes, related in some way
  - The classes in a package may or may not be related by inheritance
- The `System` class, for example, is in package `java.lang`



Some Java API Packages

java.applet	Applet classes
java.awt	Graphics, window and GUI classes
java.awt.datatransfer	Data transfer (e.g., cut-and-paste)classes
java.awt.event	Event processing classes and interfaces
java.awt.image	Image processing classes
java.awt.peer	Platform independent GUI interfaces
java.beans	JavaBeans component model API
java.io	Various types of input and output classes
java.lang	Core language classes
java.lang.reflect	Reflection API classes
java.math	Arbitrary precision arithmetic

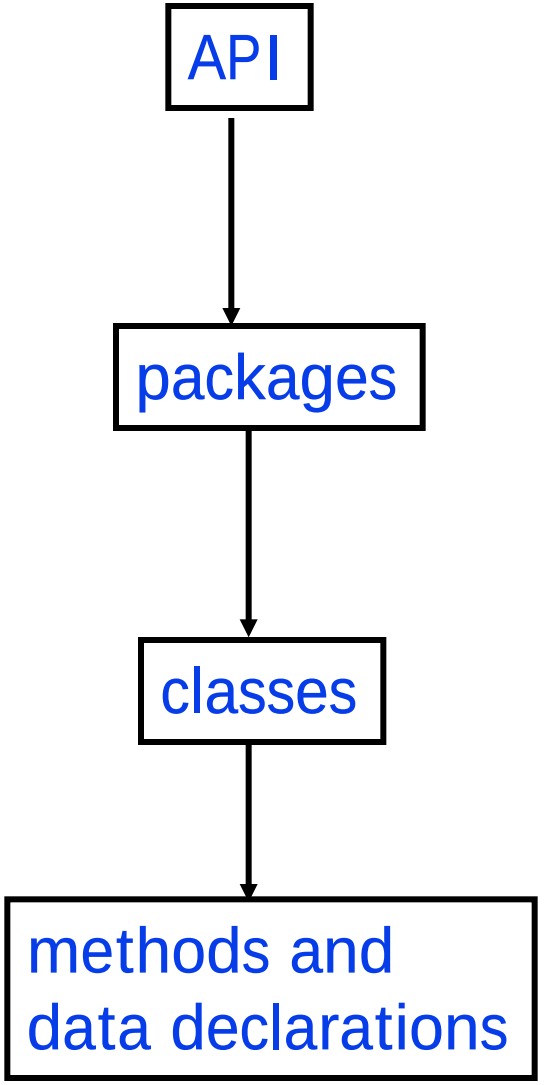
java.net	Networking classes
java.rmi	Remote Method Invocation classes
java.rmi.dgc	RMI-related classes
java.rmi.registry	RMI-related classes
java.rmi.server	RMI-related classes
java.security	Security classes
java.security.acl	Security-related classes
java.security.interfaces	Security-related classes
java.sql	JDBC SQL API for database access
java.text	Internationalization classes
java.util	Various useful data types
java.util.zip	Compression and decompression classes

How do I Create a Java Package?

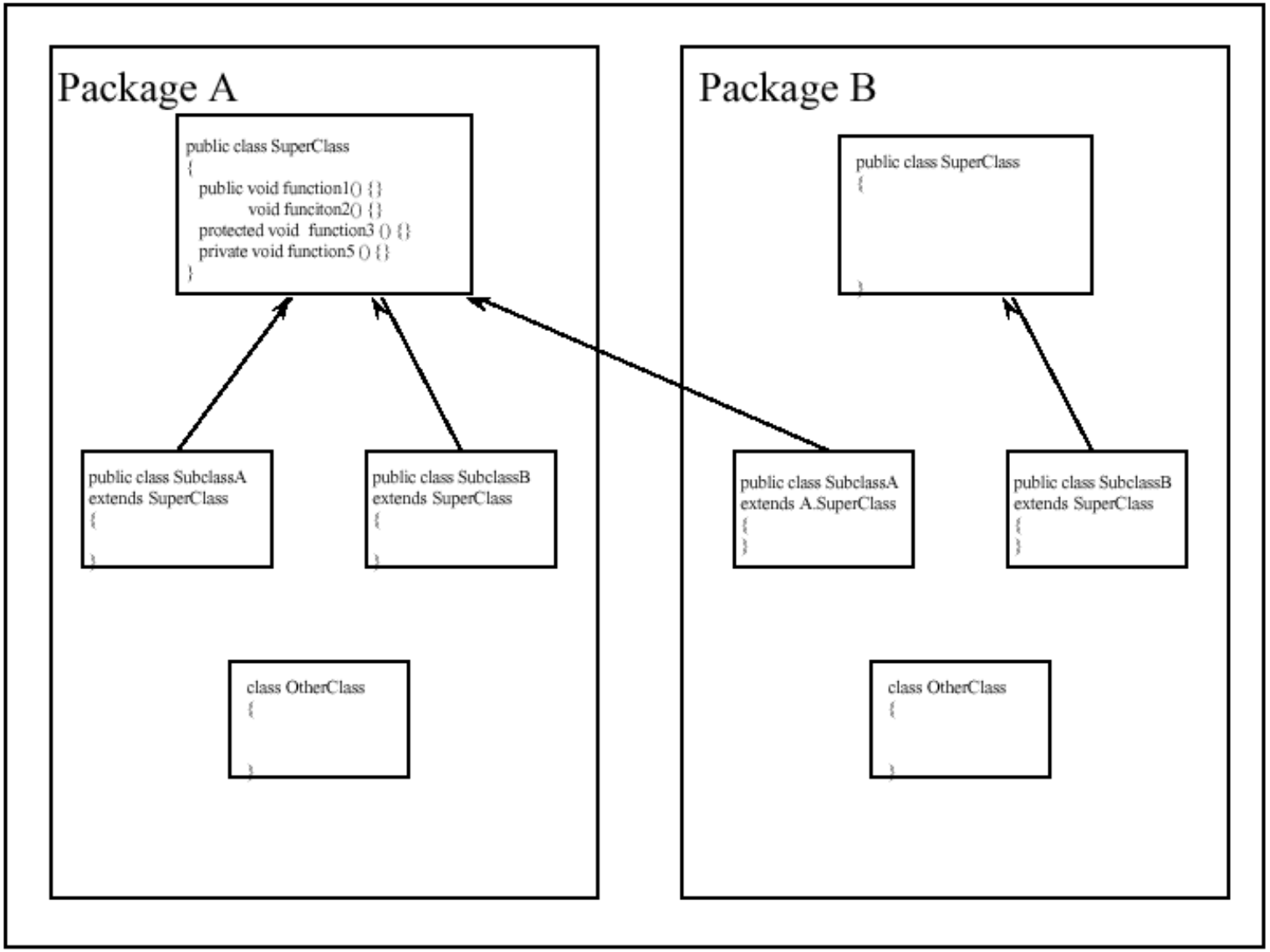
- A programmer can define a package and add classes to it
- To group several classes into a package we must:
  - Specify that all classes defined in a source file belong to a particular package using the **package statement** and
  - Put all class files under a directory with the name of the package (should be used in our CLASSPATH shell variable)
- The syntax of the package statement is:

  `package` *`package-name`*`;`

  It must be located at the top of a file, and there can be only one package statement per file

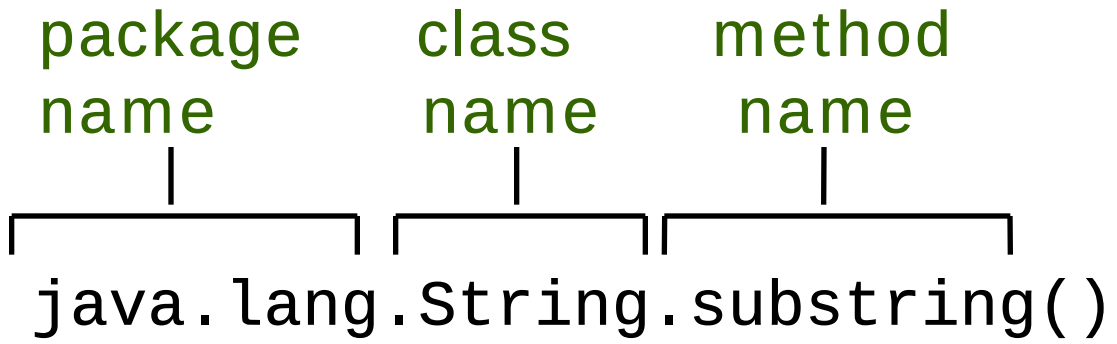


Java Packages Example



Java Globally Unique Package Names

- Proposed Internet-wide unique package naming scheme produces qualified names based on Internet domain name of the organization where the package was developed



- Using a class from the Java API can be accomplished by using its fully qualified name:

```
java.lang.System.out.println ();
```

Importing Java Packages

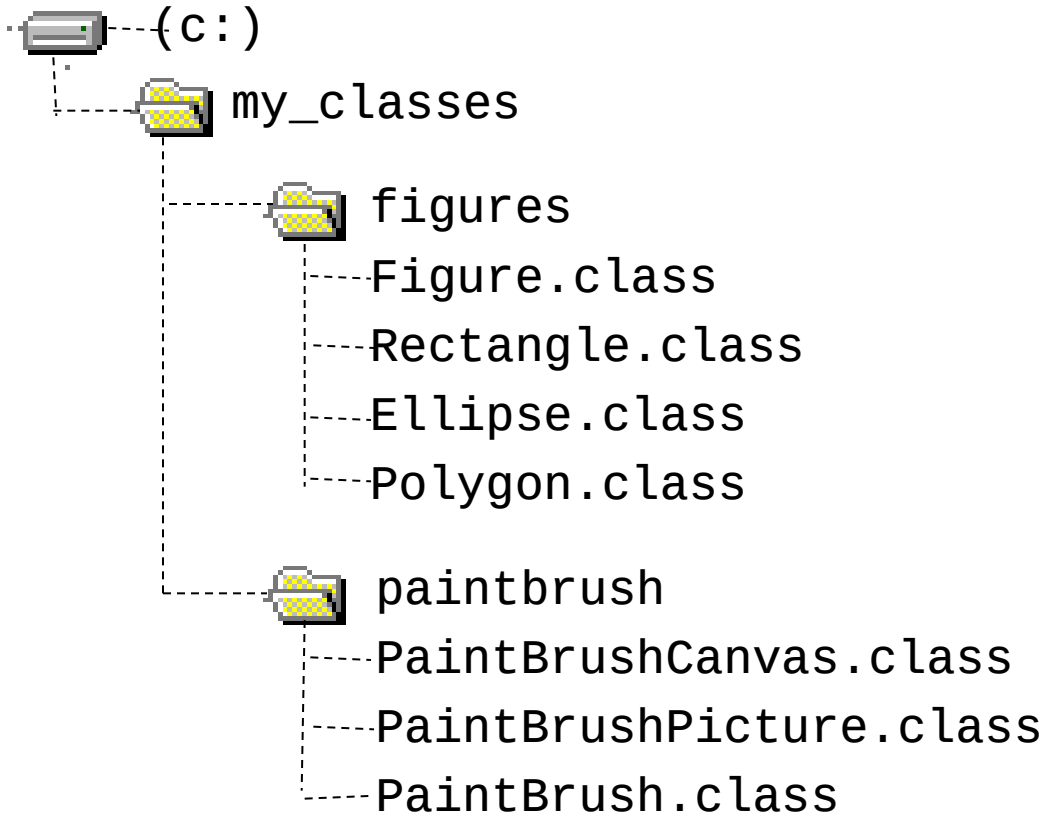
- The Java API is composed of multiple packages
  - The `import` statement is used to assert that a particular program will use classes from a particular package
- The import statement specifies particular classes, or an entire package of classes, that can be used in that program
  - Import statements are not necessary; a class can always be referenced by its fully qualified name in-line
  - If two classes from two packages have the same name and are used in the same program, they must be referenced by their fully qualified name
- The `import` *statement* has two forms:

```
import java.applet.*;
import java.util.Random;
```

  - The `java.lang` package is automatically imported into every Java program

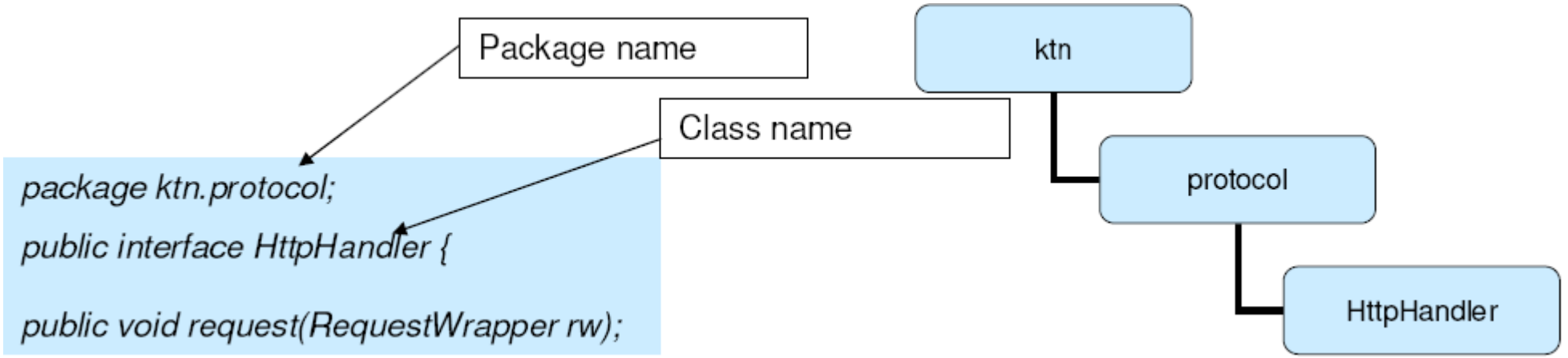
Rules for Importing Packages

- As a rule of thumb, if you will use only one class from a package, import that class specifically
- If two or more classes will be used, use the * wildcard character in the import statement to provide access to all classes in the package



Classes, Files and Packages

- Java source code is stored in files with .java extension
  - One Class = One File
  - File Name = Class Name.java (case sensitive!)
- Classes can be grouped in hierarchical packages
  - One Package = One Directory, Package Name = Directory Name
  - Sub-Package = Sub Directory
- Full name of a class is: packageName.ClassName



How do I Create an Applet?

- Create a source file named `HelloWorld.java`

```
import java.applet.Applet;
import java.awt.*;

public class HelloWorld extends
  Applet
{ public void paint(Graphics g)
   {
     g.drawString( “Hello
      World”, 20, 20);
   }
}
```

- Compiling the Applet
  - Same as compiling an Application

    ```
    > javac
      HelloWorld.java
    ```
  - Creates the class file
    - `HelloWorld.class`

How do I Run an Applet?

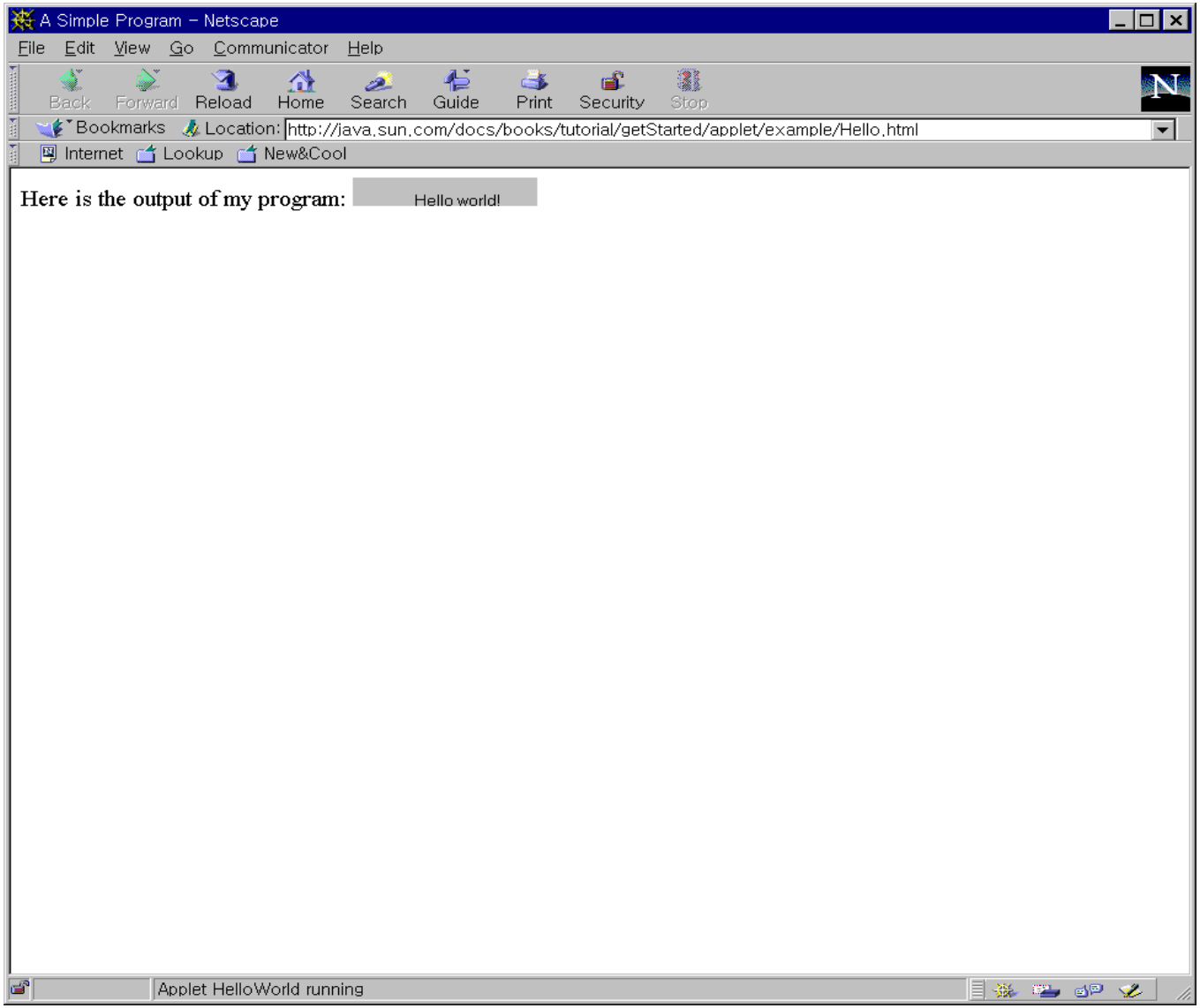
- Creating the HTML file
  - An applet runs inside a web browser so code needs to be inserted into an HTML file to view the applet
  - APPLET tag

```
<APPLET
 code="HelloWorld.class"
 WIDTH=300 HEIGHT=200>
</APPLET>
```

  - The HTML file may have any name as long as it ends with `.html` or `.htm` (e.g., `HelloWorld.html`)
- Viewing the Applet
  - Make sure the `.class` file is in the same directory as the HTML file
  - View the HTML file using a web browser such as Mozilla Firefox or Microsoft Internet Explorer
  - The JDK also provides an application for viewing applets called `appletviewer`

```
> appletviewer
  HelloWorld.html
```

Running an Applet



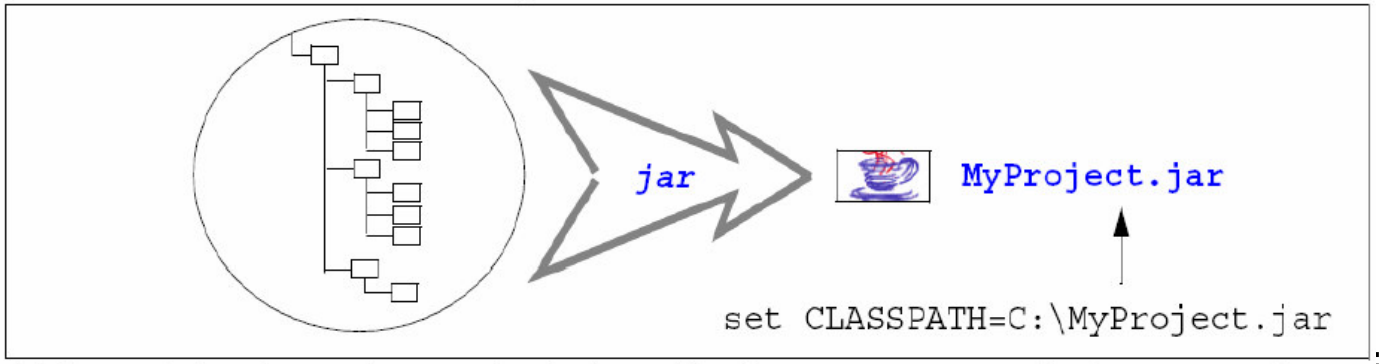
The <APPLET> Tag

```
< APPLET
      [CODEBAE = codebaseURL]
      CODE = appletFile
      [ALT = alternateText]
      [NAME = appletInstanceName]
      WIDTH = pixels
      HEIGHT = pixels
      [ALIGN = alignment]
      [VSPACE = pixels]
      [HSPACE = pixels]
>
[<PARAM  NAME = appletParameter1  VALUE = value>]
[<PARAM  NAME = appletParameter1  VALUE = value>]
……
[alternamteHTML]
</APPLET>
```

Java Applets vs. Applications

Application	Applet	
☑	☑	Compile using javac
☒	☑	Run in Web browser
☒	☑	Run in appletviewer
☑	☒	Run using java interpreter (java)
☑	☒	Start at main( ) method
☒	☑	Start with init( ), start( ), etc.

JAR and Manifest Files



- Since real applications have many class files in a package, a directory is a clumsy way to reach them
- On the web, too many files adds to download time
  - JAR contains java classes but also all other resources that you need (e.g., Images, sounds, data etc.)
- A Jar is a zip file with one additional member
  - It has a manifest file (extension *.mf)
  - This allows methods to efficiently search for classes in a jar file
- To view a jar just open it in with an un-zipper (e.g. WinZip)
  - It also can be built in WinZip if one builds the manifest file
- It is best to use packages and jars together
- Java understands how to look for packages in a jar file

JAR Files

- Java allows multiple files to be stored in one file called a Java Archive (JAR)
- A JAR file uses the same format as ZIP
- JAR files are created with the **`jar`** tool
  - `jar cf jarfile inputfile(s)`
  - **`c`** - create a new JAR file
  - **`f`** - send output to a file
- Example

```
jar cf TicTacToe.jar
 TicTacToe.class audio images
```

  - Will store the `TicTacToe.class` and all the files in the `audio` and `images` directories in the `TicTacToe.jar` file

- Running a JAR file
  - The main class can be specified in the JAR file

```
java -jar jar-file
java -jar TicTacToe.jar
```

  - Applet

```
<APPLET
  code="HelloWorld.class"
  archive="HelloWorld.jar"
  WIDTH=300 HEIGHT=200>
</APPLET>
```

Manifest Files

- JAR files can contain information about the contents of the JAR file in a manifest file
- A major use of the manifest is to specify the main class
- The main class is specified in a text file:
  - `Main-Class: HelloWorld`
- Manifest information can be incorporated into the JAR file using the **`m`** option
  - `jar cmf manifest-addition jarfile inputfile(s)`
- Example:
  - `jar cmf HelloWorld.mf HelloWorld.jar HelloWorld.java`
- Viewing the contents of a JAR file
  - use the **`t`** option to list the table of contents of the JAR file
  - `jar tf jar-file`
  - `jar tf HelloWorld.jar`
- Extracting the contents of a JAR file
  - the **`x`** option is used to extract files
  - `jar xf jar-file [archived-file(s)]`
  - `jar xf HelloWorld.jar META-INF/MANIFEST.MF`
  - The manifest file is stored in the `META-INF` directory and is called `MANIFEST.MF`

Java Resources

- Development Environments
  - Sun Software Development Kit (S2JDK)
    - Free, command line based
    - available for Solaris, Linux, Windows (NT/9X/2000/XP)
  - Borland JBuilder
    - Personal version for free
    - Visual
  - Sun Eclipse
    - free
    - Visual
  - Microsoft Visual J++
    - Not known for compatibility, not visual
  - Others…
- Documentation
  - Java API Documentation
    - All classes that ship with Java
    - Class descriptions
    - Attributes
    - Methods
  - Java Language Reference
  - The Java Tutorial
    - java.sun.com/docs/books/tutorial/

Java Platforms History

- January 1996: first official release JDK 1.0
  - Web: applets, security, URL, networking
  - GUI: Abstract Windows Toolkit (AWT)
- February 1997: JDK 1.1
  - Authentication: digital signatures, certificates
  - Distributed computing: RMI, object serialization, Java IDL/CORBA
  - Database connectivity: JDBC
  - Component architecture: JavaBean
- December 1998: Java 2 Platform (JDK 1.2)
  - Standard (J2SE), Enterprise (J2EE), and Micro (J2ME) Editions
  - JCF: Swing, Java2D, Java3D
  - Java Cryptography Extension (JCE)
  - Enterprise computing: enterprise JavaBean (EJB), servlets, Java server page (JSP), Jini, XML
  - JNI: java native interface
  - Java Multimedia Framework (JMF)
  - Embedded systems: KVM, JavaCard
  - Performance enhancement: JIT, HotSpot VM
- May 2000: J2SE v1.3
  - Enhancements in features, performance, and security: Reflection, Applet caching/jar indexing

Java Platforms History

- May 2002: J2SE v1.4
  - Numerous enhancements: I/O, assertions, regular expression, logging, etc.
  - NIO: New IO API
  - XML, XSLT, JCE, etc. all included in J2SE
  - Better Swing performance
  - Assertion facility
  - AWT: full-screen mode, headless support, better focus architecture
- 2004: J2SDK v1.5
  - Generics: Polymorphism and compile-time type safety (JSR 14)
  - Enhanced for Loop: For iterating over collections and arrays (JSR 201)
  - Autoboxing/Unboxing: Automatic conversion between primitive, wrapper types (JSR 201)
  - Typesafe Enums: Enumerated types with arbitrary methods and fields (JSR 201)
  - Varargs: Puts argument lists into an array; variable-length argument lists
  - Static Import: Avoid qualifying static members with class names (JSR 201)
  - Annotations (Metadata): Enables tools to generate code from annotations (JSR 175)
  - Concurrency utility library, led by Doug Lea (JSR-166)

Program Development in Java

Step 1: Make sure you Understand the Problem

- Example: Too many did not know when the program should terminate

- Reason:

  1. Did not understand the problem
  Solution: Speak with the client

  2. Did not carefully read the instructions
  Solution: Read!

"*If you cannot find a corresponding entry in the program then the program terminates*"

Step 2: Develop your Software

- Analysis: The process of identifying requirements
- Design: The process of developing a solution to the user's needs and requirements
- Implementation: Coding the design in a computer language such as Java
- Test: Ensuring that the finished software satisfies the requirement
- Maintenance: Adding new features and keeping the software up to date with its environment

Step 3: Develop your Program in small Increments

- Test Your Program at each stage
  - Just because it compiles does not mean it works
- Printout everything - print statements can be taken out later
  - See if the output matches what you should get
- It is much harder to fix bugs in a big program than it is in a smaller program

Step 4: Testing and Debugging

- Testing: The process of running a program with the intention of finding flaws in the code
  - Too many people run their programs to “see output” rather than to “see if the output is correct!”

- How do you Test? Prepare some test cases:
  - At the beginning, test by giving your program valid situations -Your code should be able at least to handle the “easy” cases. Make sure to test more than 1 case
  - Next provide input that will guarantee that you will run every line of code you wrote
  - Test boundary conditions. Provide input values that are valid but are on the edge between valid and invalid input
  - Test out of bounds. Provide invalid input values and see how your program reacts
  - Are those reactions valid? You may have to talk to the client

Step 4: Testing and Debugging

- Debugging: The process of fixing the errors generated by testing
  - Compile errors - do not rank to standard of a "bug"
  - "Bug" = logical Error
- Logical Errors - two kinds:
  - Type 1:On a given input the program crashes. This is a GOOD THING because you know there is a problem, and usually can tell where

```
C:\Java>java -cp com TuringMachine dat\add.tm 111011
Exception in thread "main" java.lang.NullPointerException
        at TuringMachine.trace(TuringMachine.java:26)
        at TuringMachine.run(TuringMachine.java, Compiled Code)
        at TuringMachine.main(TuringMachine.java:55)
```

  - Type 2: Invalid output - much harder to track down. If you develop your program in stages, and test each stage before going on, you can narrow your error search (with some confidence) to the new code

```
C:\Java>java -cp com TuringMachine dat\add.tm 111011
111011
C:\Java>_
```

Be open to the possibility that a previously undetected error has been located!

Step 4: Testing and Debugging

- How do you fix the bugs? By the process of elimination
  - If you get an error message, then read the error message

```
C:\Java>java -cp com TuringMachine dat\add.tm 111011
Exception in thread "main" java.lang.NullPointerException
        at TuringMachine.trace(TuringMachine.java:26)
        at TuringMachine.run(TuringMachine.java, Compiled Code)
        at TuringMachine.main(TuringMachine.java:55)
```

The method where the problem originated

methods that led to the problem

- Use `println` statements

```
public sometype problemMethod(parameters){
     code line1
     System.out.println("1"+vari
     code line2
     System.out.println("2"+vari
     code line3
     System.out.println("3"+vari
     …
}
```

Start by placing a few *println* statements
Based on the output of those, add more around where you suspect the problem lies until you isolate the buggy line
It is important to keep the input values the same for each run till the bug is found and fixed

Common Errors for the Beginning Programmer

```
C:\Java>java -cp com TuringMachine dat\add.tm 111011
Exception in thread "main" java.lang.
                        NoClassDefFoundError: TuringMachine
```

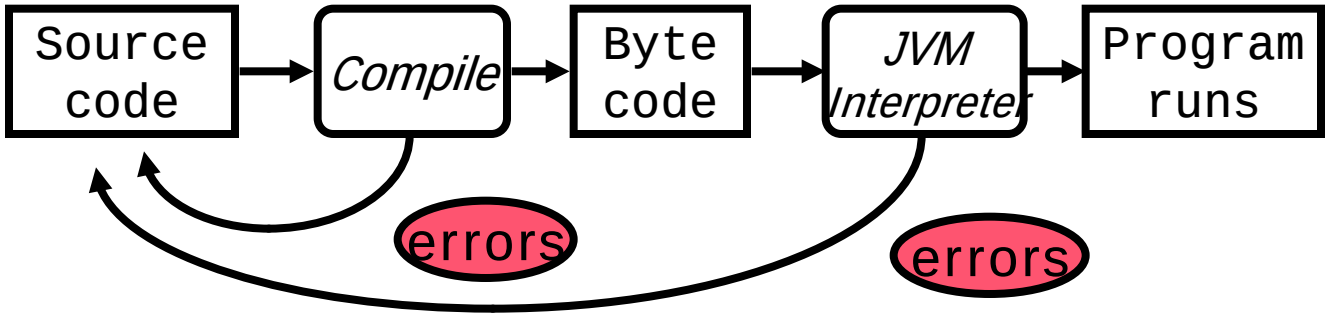
- What does this mean?

- The syntax rules of a language define how we can put symbols, reserved words, and identifiers together to make a valid program
- The semantics of a program statement define what that statement means (its purpose or role in a program)
- A program that is syntactically correct is not necessarily logically (semantically) correct
- A program will always do what we tell it to do, not what we meant to tell it to do

Handling Program Errors in Java

Errors

- A program can have three types of errors:
  - **Syntax and semantic errors**: occur during program compilation and an executable version of the program is not created
  - **Run-time errors**: occur during program execution and cause abnormal program termination
  - **Logical errors**: occur during program execution and produce incorrect results due to problems in algorithm or design or translation of algorithm to code!



Exceptions: Why?

- Error Handling: So far
  - have done very little error handling
  - have assumed that things will work as intended

- Rules of Thumb:
  - programmers must test and debug to find and correct compile-time errors
  - compiler doesn't solve the problem of run-time errors (e.g., incorrect values or state)
  - programmer must insure that run-time errors result in "graceful" program behavior
  - "graceful" means "the program doesn't just produce wrong effects or blow up!

Exceptions: Traditional Methods

- In traditional procedural programming languages, there were various ways of handling run-time errors

- One way is to return some kind of value indicating whether the procedure succeeded or not

- For example:

```
public boolean someMethod( )
{
   // if this method suceeded, return true
   // otherwise return false
}
```

Note the blurring of the logical distinction between *procedures* and *functions*… a bad engineering habit!

Exceptions: Traditional Methods

- Traditional means of error handling can quickly become ugly, complex and unmanageable.

- For example: If we *intend* to do the sequence of calling three procedures followed by one or more instructions, e.g.,

```
someMethod( );

someOtherMethod( );

someThirdMethod( );

/* do some intended actions*/
```

we can find ourselves with code that looks like . . .

Exceptions: Traditional Methods

```
if (someMethod( ) == true) {
   if (someOtherMethod( ) == true)  {
      if (someThirdMethod( ) == true)  {
            //have not encountered errors;do intended actions
         }
      else {
            //handle some error caused by someThirdMethod( )
         }
    }
   else {
         //handle some error caused by someOtherMethod( )
    }
}
else {
      //handle some error caused by someMethod( )
}
```

Exceptions: Global Variables

- Another way error handling is to have the value of a global variable representing the error

```
int iErrorValue = 0;

public void someMethod( ) {
   // do someMethod's stuff here
   // if there is an error, then set iErrorValue = 1
}

public void someOtherMethod( )   {
   // do someOtherMethod's stuff here
   // if there is an error, then set iErrorValue = 2
}

public void someThirdMethod( )   {
   // do someThirdMethod's stuff here
   // if there is an error, then set iErrorValue = 3
}
```

Exceptions: Global Variables

```
public void doIt()
 {
  someMethod();
  someOtherMethod();
  someLastMethod();

  if (iErrorValue == 1)
        ...
  if (iErrorValue == 2)
        ...
  if (iErrorValue == 3)
        ...
   }
```

But: What if the run-time error stopped us from continuing?

For example: What if *`someMethod( )`* failed in such a way that we cannot go on to *`someOtherMethod( )`*?

To cope, we find ourselves with code that's nearly as messy as the earlier example which featured multiple *nested-ifs:*

Exceptions: Global Variables

```
public void doit( ) {
   someMethod( );
   if (iErrorValue == 1)  {
         ...
    } // if
   else  {
        someOtherMethod( );
        if (iErrorValue == 2)  {
              ...
        }  // if
        else  {
             someThirdMethod( );
             if (iErrorValue == 3)  {
                 …
             }  // if
             else {
                 do intended actions
             } // else
         } // else
     }// else
```

Note: with this technique we potentially must wrap **the ENTIRE program** in a series of if/else clauses, duplicating code in places

Do we prefer robustness or clarity/maintainability?

Exceptions: The Real Problem

- Both of these traditional approaches boil down to a case of the programmer simply ignoring the real problem, which is:
  - When a run-time error occurs in a method,
    - how can we stop the method without allowing it to do any damage?
    - how can we take appropriate actions to handle the error without having the program simply blow up or do something else that's bad?

- It is not acceptable for programs to fail or to do "bad behavior"!
  - Safety critical programs
  - Customer satisfaction

- We require some mechanism to recover from unexpected or abnormal run-time situations

Exceptions and Exception Handling

- *Exception*: "*a run-time event that may cause a method to fail or to execute incorrectly*"
- *Exception Handling:* the modern programming concept for dealing with run-time errors (Errors can be dealt with at place error occurs)
  - Easy to see if proper error checking implemented
  - Harder to read application itself and see how code works
- *Purpose of Exception Handling*: "*to allow graceful handling of and recovery from run-time errors*"
  - Java removes error handling code from "main line" of program
  - Not optimized, can harm program performance
  - Common handling of similar errors

When Exception Handling Should Be Used

- Exceptions should be used for
  - Processing exceptional situations
  - Processing exceptions for components that cannot handle them directly
  - Large projects that require uniform error processing
- Exceptions must not be used for
  - Errors that can be easily avoided (e.g. with a simple if)
  - Program flow control
- Common examples in Java:
  - `NullPointerException`
  - `ArithmeticException`
  - `ArrayIndexOutOfBoundsException`
- Java API contains over two dozen exceptions provided by Java

Exceptions: Terminology

Exception Terminology:

- When an exceptional condition occurs, an exception is *"thrown"* (i.e., the exception has been *recognized*)
- The flow of control is transferred to the point where the exception is *"caught"* (I.e., where the exception-handling code *responds* to it)

- In the jargon of some other programming languages, when an exception is recognized, an exception is:
  - *"raised" (in place of "thrown")*, then
  - *"handled"* (in place of *"caught"*)

- Same ideas, different jargon

General Structure of Java's Exception Handling

```
try  {
   // here goes the code that attempts to perform the
   // intended action, but that could throw an exception
   ...
}  // try
catch (ExceptionType1 e) {
   // here goes the code to handle exception type 1
   ...
} // catch Type1
catch (ExceptionType2 e) {
   // here goes the code to handle exception type 2
   ...
} // catch Type2
```

One is usually
more interested in
the type of the exception
than in manipulating it as an object,
so "e" is just an object often thrown away.

Syntax of Exception Handling Blocks

- Enclose code that may have an error in **`try`** block
- Follow with one or more **`catch`** blocks
  - Each **`catch`** block has an exception handler
- If exception occurs and matches parameter in **`catch`** block
  - Code in catch block executed
- If no exception thrown
  - Exception handling code skipped
  - Control resumes after **`catch`** blocks

Exceptions: Simple Example

```
int   iDivisor;
int   iDividend;
float fResult;

  try
  {
       // get input for divisor and dividend
       ...
      fResult = (float) iDividend / iDivisor;
      System.out.println(fResult);
  }
  catch (ArithmeticException e)
  {
     System.out.println("The divisor was 0");
     ...
}
```

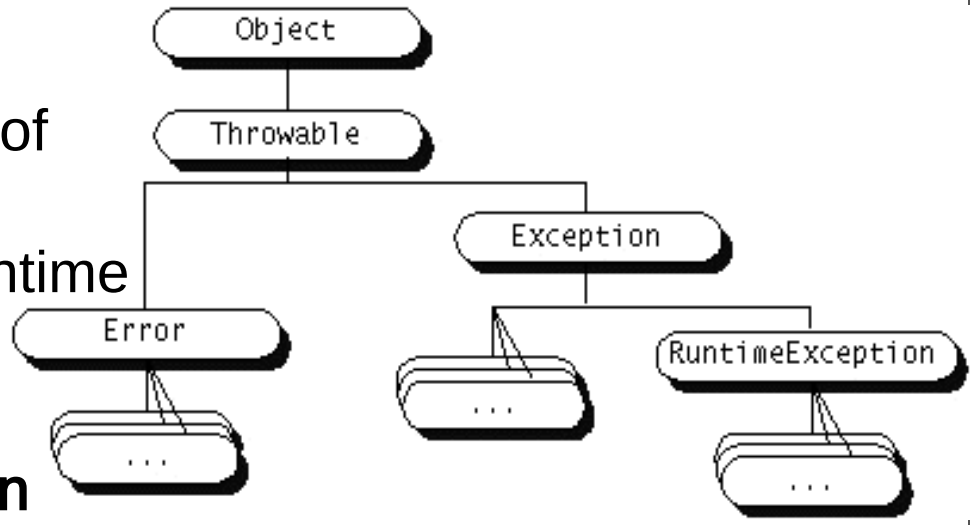
See, we don't
Care about the exception,
Just about its type being arithmetic error

Exception Types

- Errors
  - Notifying for critical events
  - Generally not to be caught and handled
- Checked exceptions
  - Must be listed in **`throws`** clause of method
  - All exceptions that are neither runtime exceptions or errors
- Run-time exceptions
  - Derive from **`RunTimeException`**
  - Some exceptions can occur at any point

    `ArrayIndexOutOfBoundsException`

    `NullPointerException`
  - Most avoidable by writing proper code



Exception Handling: Simple Example

```
public class Sum {
  public static void main(String[] args){
      int sum=0;
      for (int i=0; i<args.length; i++){
            int number;
            try{
                number=Integer.parseInt(args[i]);
                sum+=number;
            }
            catch(NumberFormatException e){
      System.err.println(args[i]+" is not a number.");
            }
      }
      System.out.println("The sum is: "+sum);
  }
}
```

If an exception occurs, the try block is terminated and nothing is summed to the sum variable

catch contains the code notifying the user for the error

Nested Exception Handling Blocks

- Exception Handling blocks may be nested
- When an exception occurs, Java Runtime searches for the nearest matching handler for it
- Only the nearest matching handler is executed for an exception. All other handlers skipped

```
try{
   …
   try{
      …
   }
   catch ( Exception1 ){...}
   …
}
catch( Exception2 ){...}
```

Rethrowing an Exception

- Rethrowing exceptions
  - Use if handler cannot process exception
  - Rethrow exception with the statement:

    `throw e;`
    - Detected by next enclosing **`try`** block
  - Handler can always rethrow exception, even if it performed some processing

```
try{
   try{
      …
   }
   catch ( IOException e){...; throw e;}
}
catch( Exception e){...}
```

Both handlers are executed

Throws Clause

- **`throws`** clause lists the exceptions that can be thrown by a method
- Java compiler to enforces exception handling for the exceptions thrown by a method

```
int g( float h ) throws a, b, c
{
   // method body
}
```

Catch-or-declare requirement

- Let's say method A is declared to throw exception E.
- If method B calls A, then B must either:
  - Catch exception E and/or
  - Be declared to throw E as well

Using `printStackTrace` and `getMessage`

- Class **`Throwable`**
  - Superclass of all exceptions
  - Method **`printStackTrace`**
    - Prints method call stack for caught **`Exception`** object
      - Most recent method on top of stack
    - Helpful for testing/debugging

```
Exception thrown in method3
java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:28)
        at UsingExceptions.method2(UsingExceptions.java:23)
        at UsingExceptions.method1(UsingExceptions.java:18)
        at UsingExceptions.main(UsingExceptions.java:8)
```