



P.O. Box 342-01000 Thika
Email: info@mku.ac.ke
Web: www.mku.ac.ke

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: BIT3106

COURSE TITLE : OBJECT ORIENTED PROGRAMMING

Instructional Manual for BBIT – Distance Learning

E-mail:

CONTENT

Introduction to Object Orientation 3

Overview of object oriented analysis diagrams in UML 7

Object Oriented Programming Using Java21

Variables and Data types34

Using Keyboard Input 44

Control Structures50

Arrays.....63

Classes and Objects73

Java Applets99

Drawing and Filling Objects111

Creating User Interfaces with the awt133

Retrieving and Using Images151

Event Handling in Java157

Introduction to Object Orientation

Definition

Object Oriented Programming (OOP is a programming method that combines data and instructions into a self-sufficient " *object*".

"Object orientation provides a new paradigm for software construction. In this new paradigm, objects and classes are the building blocks, while methods, messages and inheritance produce the primary mechanisms".- *Ann L. Winbald, Samuel D. Edwards and David R. King in the book, "Object Oriented Software"*.

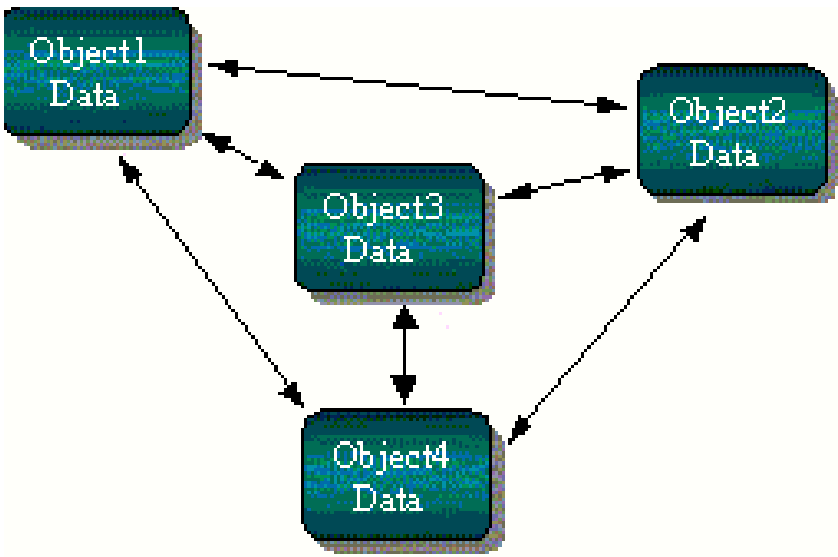
Overview

In an object-oriented program, objects represent the design. Objects have two sections, fields (instance variables) and methods.

Fields represent what an object is.

Methods represent how an object is used.

These fields and methods are closely tied to the real world characteristics and use of the object. An object is used by means of its methods. The following figure shows a view of object oriented programming



"Programming in the Small" and "Programming in the Large"

“Programming in the small”

Programs are developed by a single programmer/ small group of programmers.

All aspects of the project can be understood by a single individual.

The major problem is the development of the algorithms and data structures needed to solve the task at hand.

“Programming in the large”

The software system is developed by a large team of programmers, often with considerable specialization.

No single individual can (likely) understand all aspects of the project.

The major problem is the coordination of the diverse aspects of the project--people and software systems

Object Orientation

Assumptions:

Describing large, complex systems as interacting objects make them *easier to understand* than otherwise.

The *behaviors* of real world objects tend to be *stable* over time.

The different *kinds* of real world objects tend to be *stable*. (That is, new kinds appear slowly; old kinds disappear slowly.)

Change tend to be *localized* to a few objects.

An Object Model

Our object model includes four components:

objects(i.e., abstract data structures)

classes(i.e., abstract data types)

inheritance(hierarchical relationships among ADTs)

polymorphismby inheritance

Encapsulation

Abstraction

Objects

An *object* is a separately identifiable entity that has a set of operations and a state that records the effects of the operations. That is, an object is essentially the same as an abstract data structure as we have discussed previously.

Objects are characterized by:

state,
operations,
identity.

Classes

A *class* is a template for creating objects.

A class describes a collection of related objects (i.e., instances of the classes).

Objects of the same class have common operations and a common set of possible states.

The concept of class is closely related to the concept of abstract data type that we discussed previously.

A class description includes definitions of
operations on objects of the class,
the possible set of states.

Inheritance

- A class D *inherits* from class B if D's objects form a subset of B's objects.

Class D's objects must support all of the class B's operations (but perhaps are carried out in a special way).

Class D may support additional operations and an extended state (i.e., more information fields).

Class D is called a *subclass* or a *child* or *derived class*.

Class B is called a *superclass* or a *parent* or *baseclass*.

Polymorphism

The concept of *polymorphism* (literally "many forms")

Polymorphism appears in several forms.

Overloading (or *ad hoc polymorphism*)

Parametric polymorphism

denotes the use of parameterized type (or class) definitions.

Polymorphism by inheritance (sometimes called pure polymorphism .

The concept of *polymorphism*(literally "many forms")

Polymorphism appears in several forms.

Overloading(or *ad hoc polymorphism*)

Parametric polymorphism

denotes the use of parameterized type (or class) definitions.

Polymorphism by inheritance (sometimes called pure polymorphism .

Encapsulation

Encapsulation a way of packaging information. Encapsulation allows the programmer to present clearly specified interfaces around the services they provide. The programmer can decide what should be hidden and what is intended to be visible.

Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations. It is the art of concentrating on the essential and ignoring the non-essential. Classes that use the concept of data abstraction are known as "abstract data types". Abstract data types are achieved by making certain variables and methods in a class private.

Why is OOP Popular?

OOP is revolutionary idea, totally unlike anything that has come before programming

OOP is an evolutionary step, following naturally on the heel of earlier programming abstractions.

People hoped that it would quickly and easily lead to increased productivity and improved reliability - solve software crises

People hoped for easy transition from existing languages

Resonant similarity to techniques of thinking about problems in other domains.

Overview of object oriented analysis diagrams in UML

The purpose of UML, or Unified Modeling Language, is communication; to be specific, it is to provide a comprehensive notation for communicating the requirements, architecture, implementation, deployment, and states of a system.

UML, the Unified Modeling Language, is a graphical language designed to capture the artifacts of an OOAD process. It provides a comprehensive notation for communicating the requirements, behavior, architecture, and realization of an Object-Oriented design. UML provides you with a way to create and document a model of a system.

Object-Oriented Analysis

In software development, *analysis* is the process of studying and defining the problem to be resolved. It involves discovering the requirements that the system must perform, the underlying assumptions with which it must fit, and the criteria by which it will be judged a success or failure.

Object-Oriented Analysis (OOA), then, is the process of defining the problem in terms of objects: real-world objects with which the system must interact and candidate software objects used to explore various solution alternatives. The natural fit of programming objects to real-world objects has a big impact here: you can define all of your real-world objects in terms of their classes, attributes, and operations.

Design

If analysis means defining the problem, then *design* is the process of defining the solution. It involves defining the ways in which the system satisfies each of the requirements identified during analysis.

Object-Oriented Design (OOD), then, is the process of defining the components, interfaces, objects, classes, attributes, and operations that will satisfy the requirements. You typically start with the candidate objects defined during analysis, but add much more rigor to their definitions. Then you add or change objects as needed to refine a solution. In large systems, design usually occurs at two scales: architectural design, defining the components from which the system is composed; and component design, defining the classes and interfaces within a component.

Models

Did you ever build a model ship? When I was young, my mom and I built a model of the famous clipper ship *Cutty Sark*.³ I’ve always been fascinated by the tall ships of old; but I really learned about how they work when we built that model. All of the strange nautical terminology from the old pirate movies—forecastle, capstan, main mast, and especially belaying pins (“You mean they’re not just there so somebody can pull one out and swing it as a club?”)—gained concrete meaning when I assembled them and saw them in the context of the entire system.

Central goal of using UML in OOAD: to let you study and understand a system via a model of that system. Like aerodynamic engineers, construction architects, and others in the physical engineering world, you’ll build models of systems yet to be built, not just models of existing systems. Your models will let you explore design alternatives and test your understanding of the system at a much faster rate and much lower cost than the rate and cost associated with actually building the system.

In the case of OOAD with UML, your models consist primarily of diagrams: static diagrams that depict the structure of the system, and dynamic diagrams that depict the behavior of the system. With the dynamic diagrams, you can trace through the behavior and analyze how various scenarios play out. With the static diagrams, you can ensure that each component or class has access to the interfaces and information that it needs to carry out its responsibilities. And it’s very easy to make changes in these models: adding or moving or deleting a line takes moments; and reviewing the change in a diagram takes minutes. Contrast that with actually building the code: hours to implement the change, hours more to test and review it.

Your core artifact of the OOAD process is the model. In fact, you will likely have multiple models:

- **Analysis Model.** This is a model of the existing system, the end user’s requirements, and a high-level understanding of a *possible* solution to those requirements.
- **Architecture Model.** This is an evolving model of the structure of the solution to the requirements defined in the Analysis Model. Its primary focus is on the architecture: the components, interfaces, and structure of the solution; the deployment of that structure across nodes; and the trade-offs and decisions that led up to that structure.
- **Component (Design) Models.** This is a number of models (roughly, one per component) that depict the internal structure of the pieces of the Architecture Model. Each Component Model focuses on the detailed class structure of its component, and allows the design team to precisely specify the attributes, operations, dependencies, and behaviors of its classes.

Depending on your development process, you may have even more models: a Business Model, a Domain Model, possibly others. The major benefit of models is that you can make *model* changes far earlier in the development cycle than you can make *code* changes, and far easier. And because you can make changes early, you can make your mistakes early. And that's a good thing, because early detection and correction is cheap detection and correction. Modeling will let you catch your most costly bugs early; and early detection and correction can save you a factor of 50 to 200 on the cost and schedule of a bug fix.

Fundamentals of Object Oriented Modeling

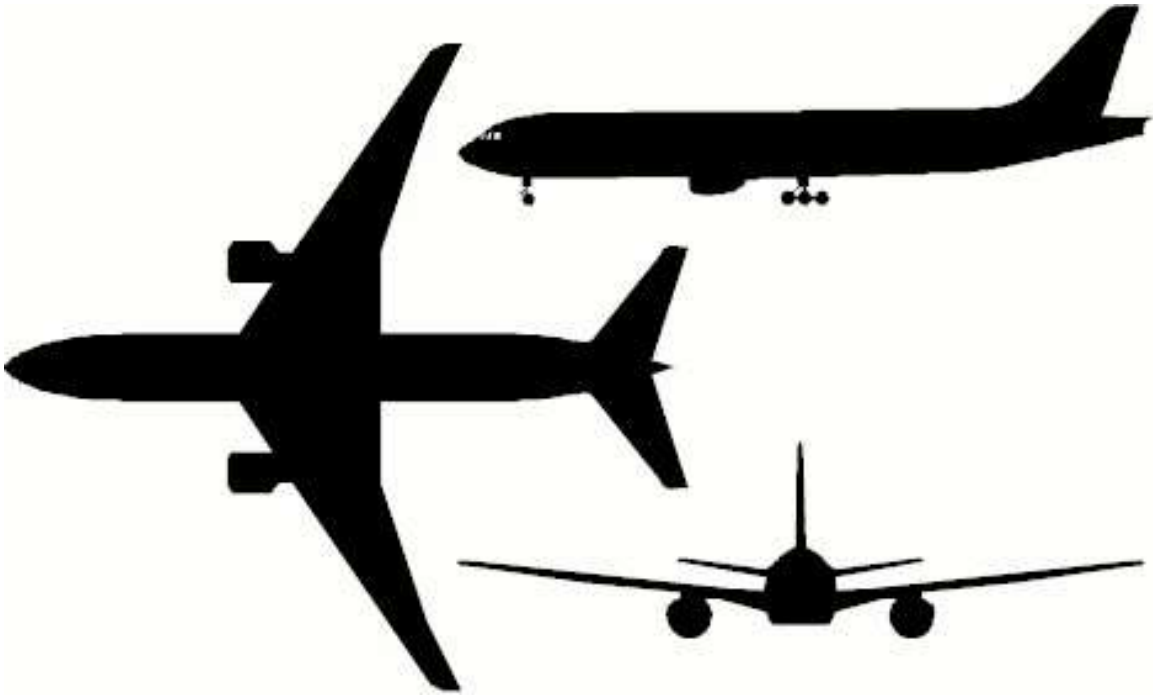
Introduction

Abstraction is a fundamental principle of modeling. A system model is created at different levels, starting at the higher levels and adding more levels with more detail as more is understood about the system. When complete, the model can be viewed at several levels. So abstraction is about:

- Looking only at the information that is relevant at the time
- Hiding details so as not to confuse the bigger picture

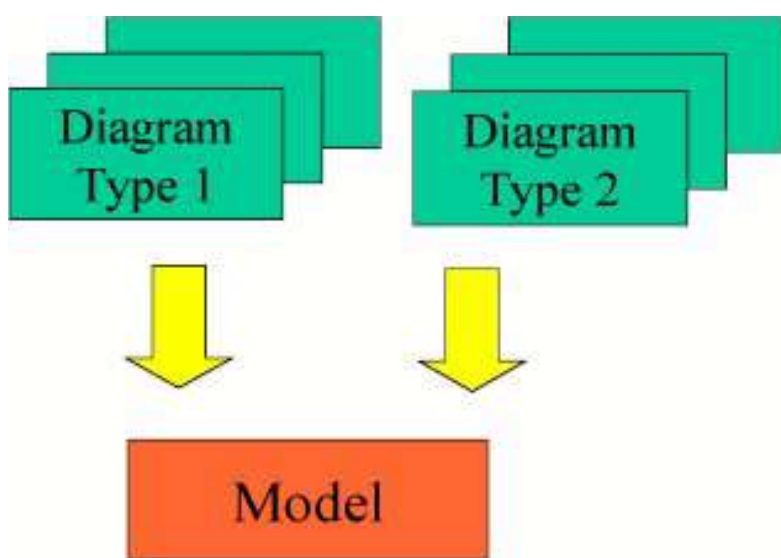
When we model, we create a number of views of the system being described - usually 2 or 3. For a complete description 3 are normally needed. Each view is an 'orthogonal' view. Each view is needed for a full understanding of the system. Views are orthogonal when:

- Each view has information that is unique to that view
- Each view has information that appears in other views
- The information that is common between views is consistent. (In this case it isn't. Can you spot the problem?)



Model Organisation

All model syntaxes provide a number of model elements which can appear on one or more diagram types. The model elements are contained with a central model, together with all their properties and connections to other model elements. The diagrams are independent views of the model, just as a number of computer screens looking into different records or parts of a database shows different views.



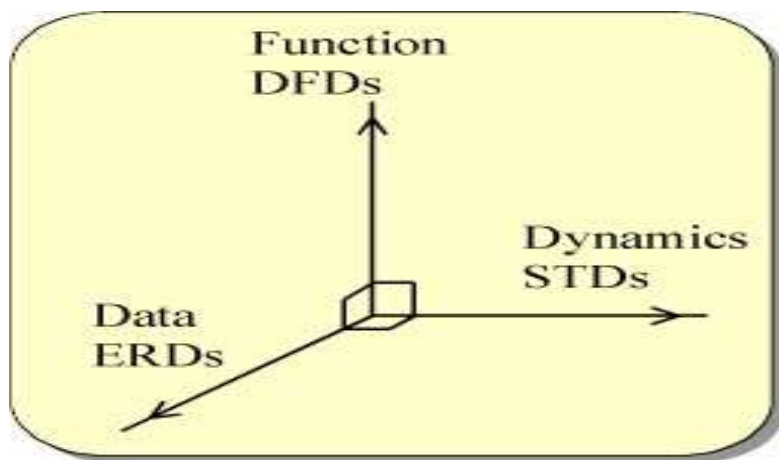
Structured Analysis

In structured analysis there are three orthogonal views:

- The functional view, made up of data flow diagrams, is the primary view of the system. It defines what is done, the flow of data between things that are done and

provides the primary structure of the solution. Changes in functionality result in changes in the software structure.

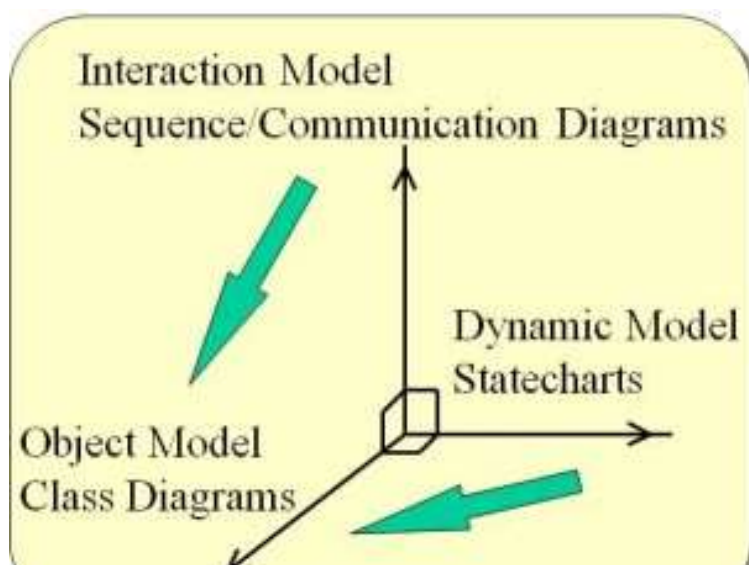
- The data view, made up of entity relationship diagrams, is a record of what is in the system, or what is outside the system that is being monitored. It is the static structural view.
- The dynamic view, made up of state transition diagrams, defines when things happen and the conditions under which they happen.



Object Orientation

Object oriented software is based on the static, or object model: the things in the system, or, the things outside the system about which we need to record information, and their relationships. This is the most stable view in the system. It is why an object oriented model produces more stable software over a long period.

- **Functionality** from the interaction model is mapped onto the object model.
- **The interaction model** is, in turn, derived from the use case model which defines the functionality of the system from a user's perspective.



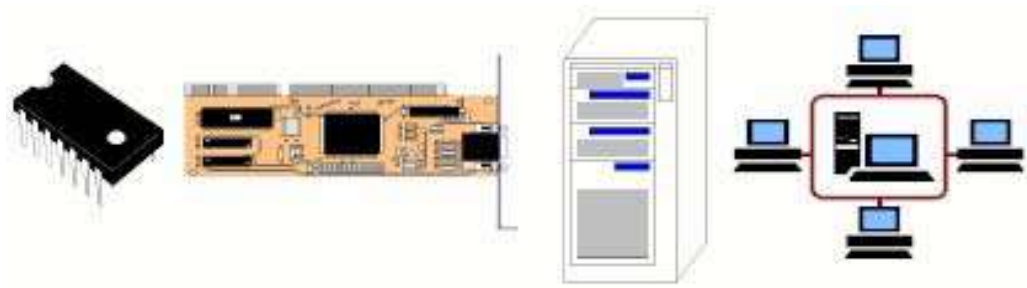
- **The dynamic model** defines the order and conditions under which things are done and is also mapped onto the object model.

Encapsulation of Hardware

The concept of encapsulation of data and functionality that belongs together is something which the hardware industry has been doing for a long time. Hardware engineers have been creating re-useable, re-configurable hardware at each level of abstraction since the early sixties. Elementary boolean functions are encapsulated together with bits and bytes of data in registers on chips. Chips are encapsulated together on circuit boards. Circuit boards are made to work together in various system boxes that make up the computer. Computers are made to work together across networks. Hardware design, therefore, is totally object oriented at every level and is, as a result, maximally re-useable, extensible and maintainable; in a single word: flexible. Applying object-orientation to software, therefore, could be seen as putting the engineering into software design that has existed in hardware design for many years.

Hardware encapsulates data and function at every level of abstraction

Maximises maintainability, reuse and extension



Encapsulation of Software

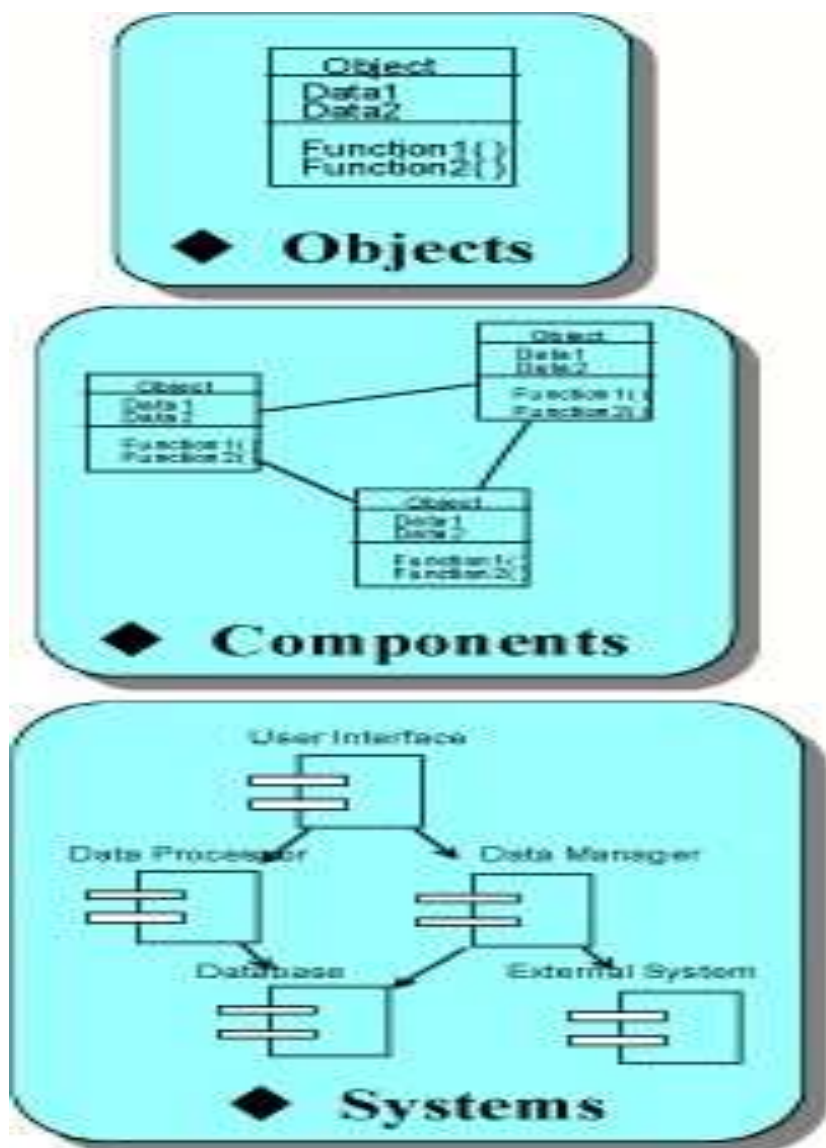
In well developed object oriented software, functionality and data is encapsulated in objects. Objects are encapsulated in components. Components are encapsulated into systems. If this is done well the result is:

Maximal coherence

Minimal interconnection

Solid interface definitions

Maximum maintainability, reuse and extensibility



Unified Modeling Language (UML)

The Unified Modeling Language was originally developed at Rational Software but is now administered by the Object Management Group (see link). It is a modeling syntax aimed primarily at creating models of software-based systems, but can be used in a number of areas.

It is:

Syntax only - UML is just a language; it tells you what model elements and diagrams are available and the rules associated with them. It does not tell you what diagrams to create.

Comprehensive - it can be used to model anything. It is designed to be user-extended to fill

any modeling requirement.

Language-independent - it doesn't matter what hi-level language is to be used in the code.

Mapping into code is a matter for the case tool you use.

Process-independent - the process by which the models are created is separate from the definition of the language. You will need a process in addition to the use of UML itself.

Tool-independent - UML leaves plenty of space for tool vendors to be creative and add value to visual modelling with UML. However, some tools will be better than others for particular applications.

Well documented - the UML notation guide is available as a reference to all the syntax available in the language.

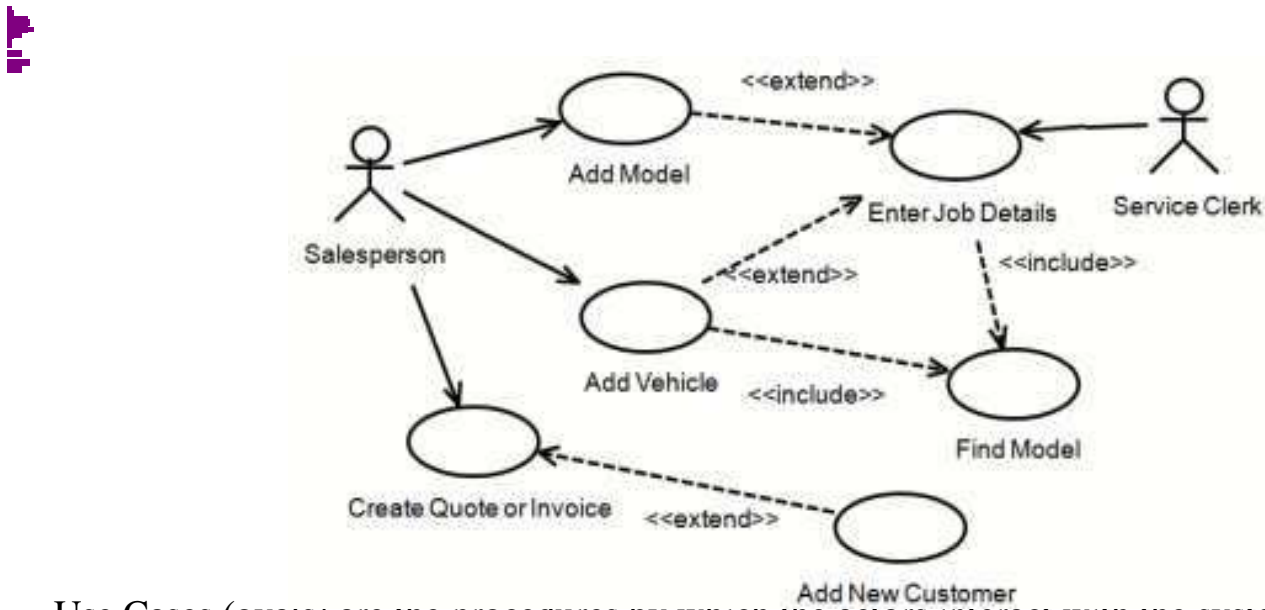
Its application is not well understood - the UML notation guide is not sufficient to teach you how to use the language. It is a generic modelling language and needs to be adapted by the user to particular applications.

Originally just for system modeling - some user-defined extensions are becoming more widely used now, for example, for business modelling and modelling the design of web-based applications.

Use case

The use case diagram shows the functionality of the system from an outside-in viewpoint.

Actors (stick men) are anything outside the system that interacts with the system.



Use Cases (ovals) are the procedures by which the actors interact with the system.

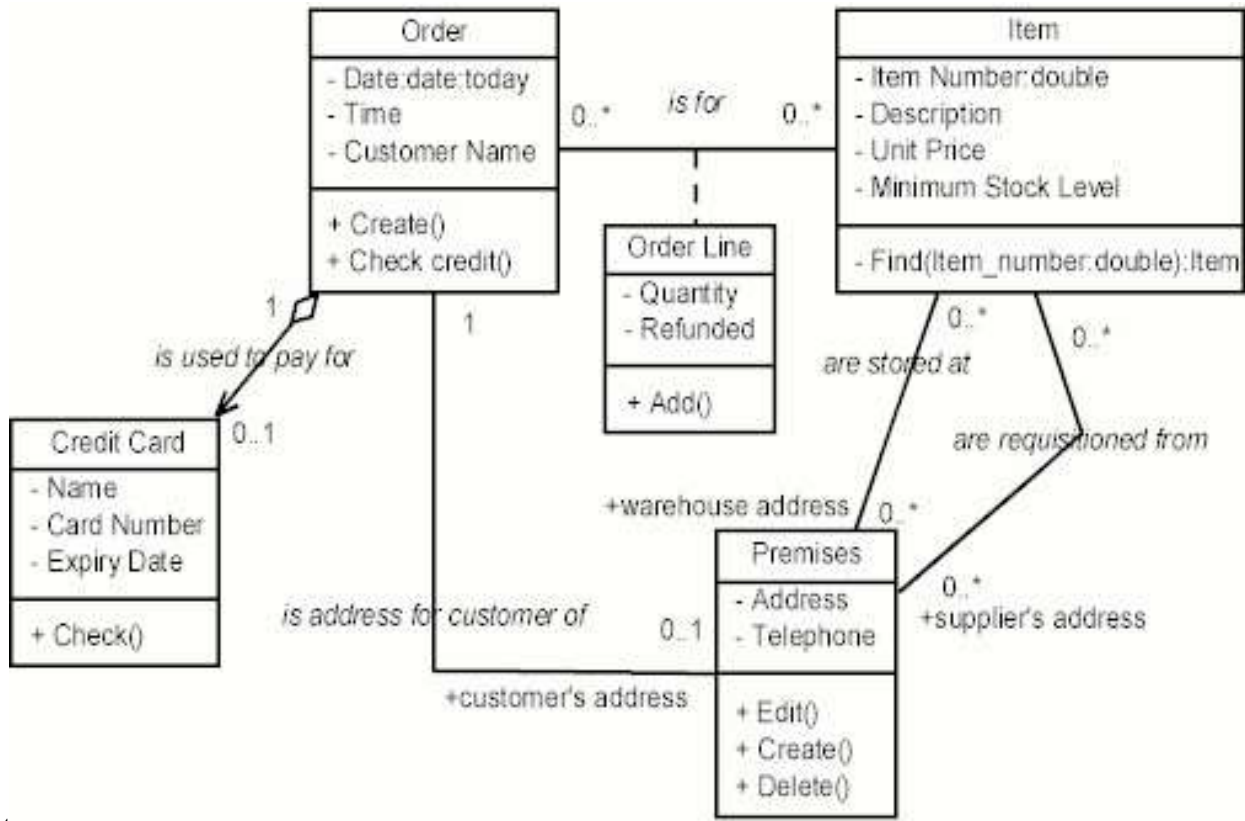
Solid lines indicate which actors interact with the system as part of which procedures.

Dashed lines show dependencies between use cases, where one use case is 'included' in or 'extends' another.

Class diagrams

Class diagrams show the static structure of the systems. Classes define the properties of the objects which belong to them. These include:

Attributes - (second container) the data properties of the classes including type, default value and constraints.



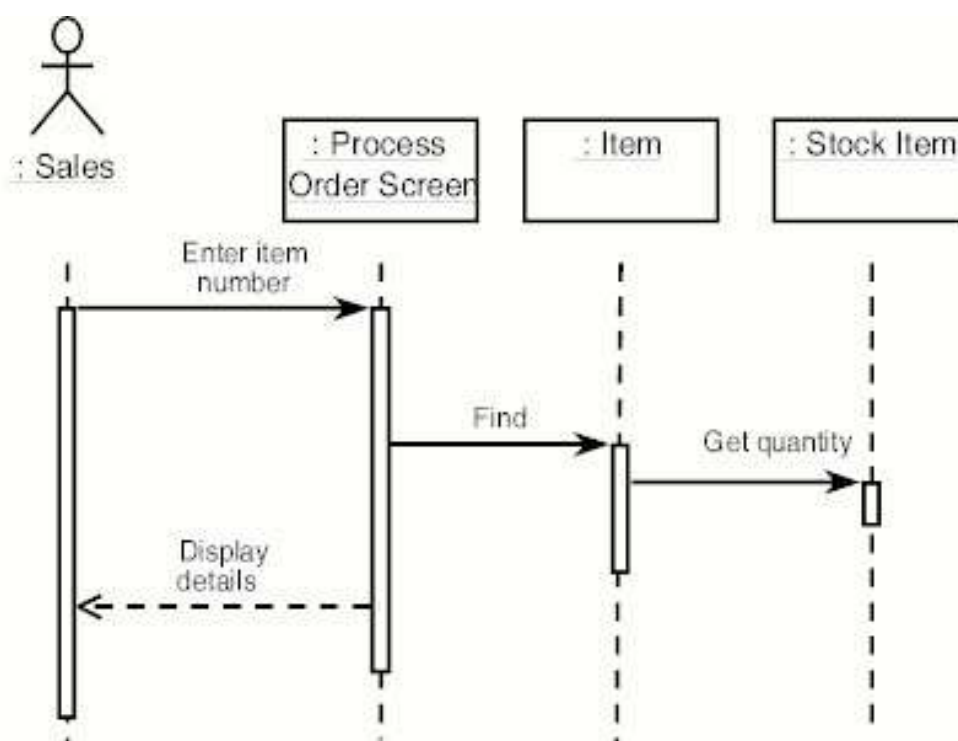
Operations - (third container) the signature or the functionality that can be applied to the objects of the classes including parameters, parameter types, parameter constraints, return types and the semantics.

Associations - (solid lines between classes) the references, contained within the objects of the classes, to other objects, enabling interaction with those objects.

Sequence Diagram

Sequence diagrams show potential interactions between objects in the system being defined. Normally these are specified as part of a use case or use case flow and show how the use case will be implemented in the system. They include:

Objects - oblong boxes or actors at the top - either named or just shown as belonging to a class, from, or to which messages are sent to other objects.



Messages - solid lines for calls and dotted lines for data returns, showing the messages that are sent between objects including the order of the messages which is from the top to the bottom of the diagram.

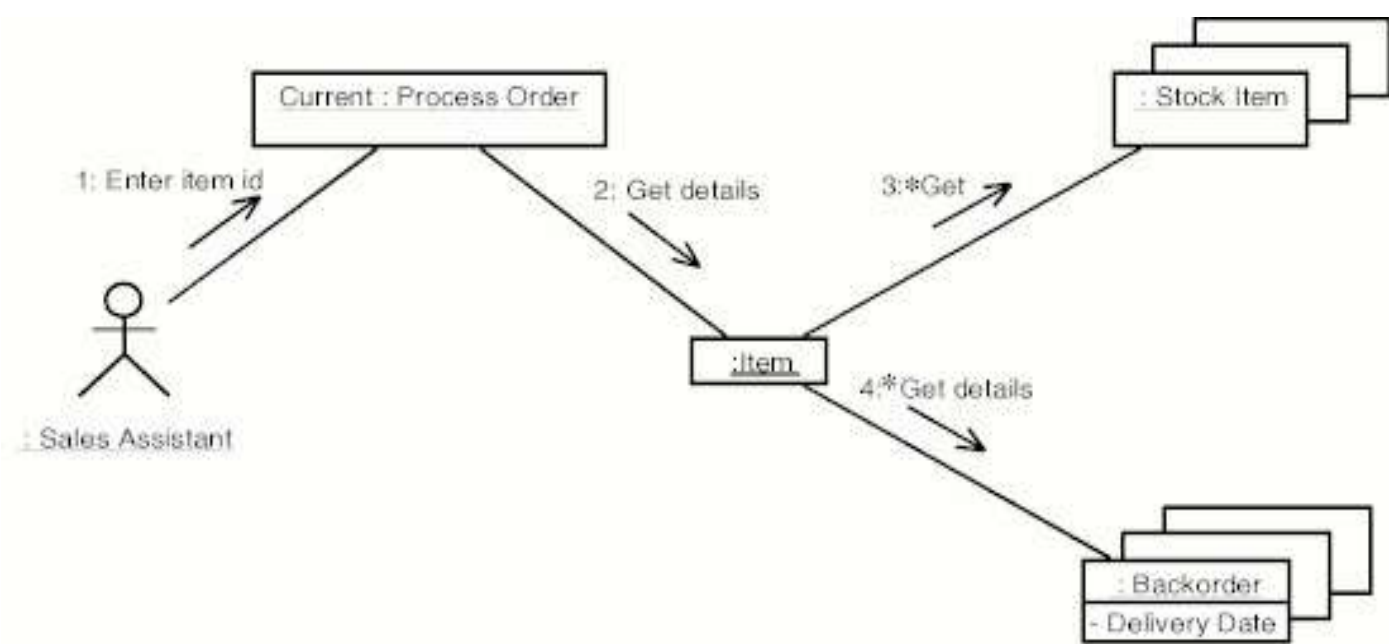
Object lifelines - dotted vertical lines showing the lifetime of the objects.

Activation - the vertical oblong boxes on the object lifelines showing the thread of control in a synchronous system.

Communication Diagrams

Communication Diagrams show similar information to sequence diagrams, except that the vertical sequence is missing. In its place are:

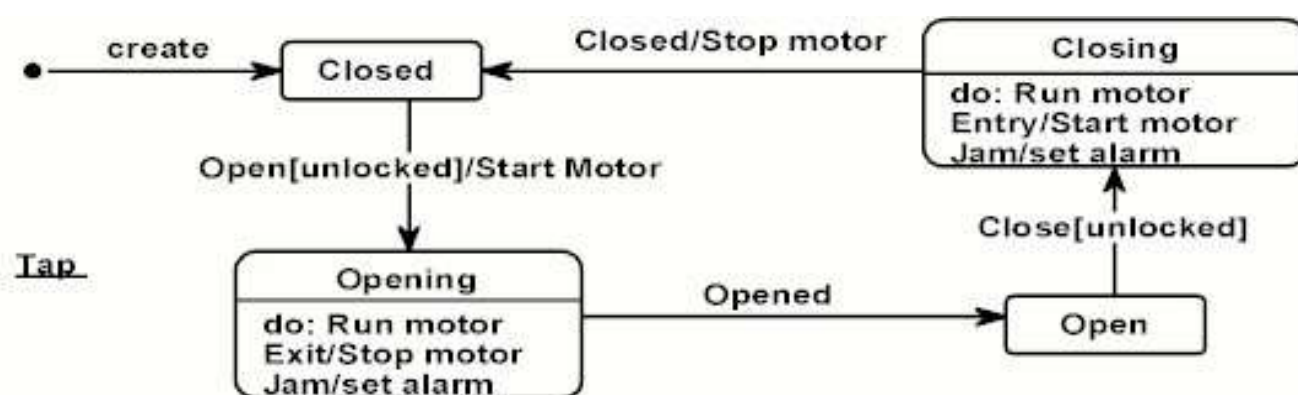
Object Links - solid lines between the objects. These represent the references between objects that are needed for them to interact and so show the static structure at object level. On the links are:



State Machine Diagram

State Machine Diagrams, or Statecharts, are often used more in real-time embedded systems than in information systems, show for a class, the order in which incoming calls to operations normally occur and the conditions under which the operations respond and the response. They are a class-centric view of system functionality, as opposed to sequence and collaboration diagrams which are a use case-centric view of functionality. They include:

States - oblong boxes which indicate the stable states of the object between events.



Transitions - the solid arrows which show possible changes of state.

Events - the text on the transitions before the '/' showing the incoming call to the object interface which causes the change of state.

Conditions - a boolean statement in square brackets which qualifies the event.

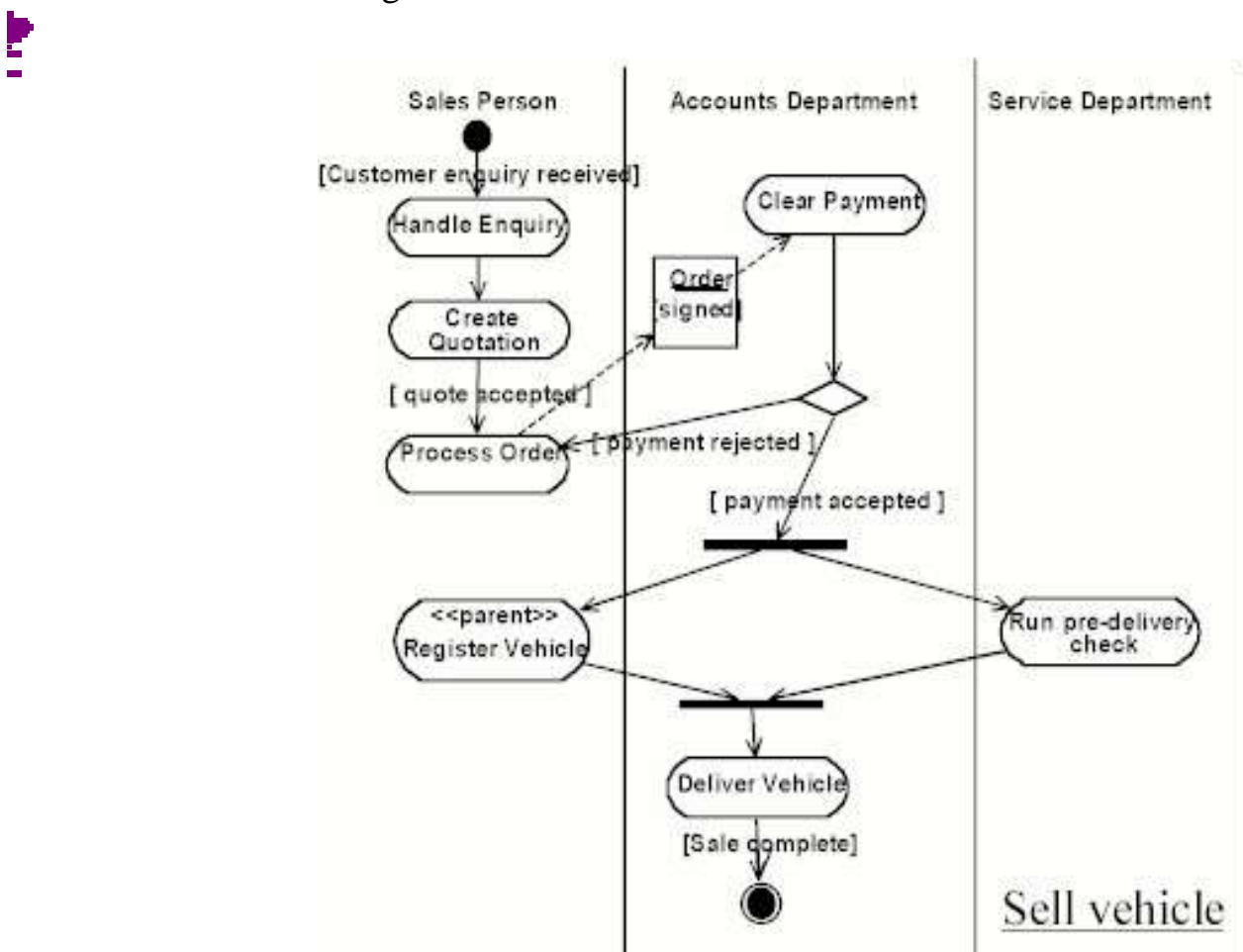
Actions - the text after the '/' which defines the objects response to the transition between states.

Extra syntax which defines state-centric functionality

Activity Diagram

A UML Activity Diagram is a general purpose flowchart with a few extras. It can be used to detail a business process, or to help define complex iteration and selection in a use case description. It includes:

Active states - oblongs with rounded corners which describe what is done.



Transitions - which show the order in which the active states occur and represent a thread of activity.

Conditions - (in square brackets) which qualify the transitions.

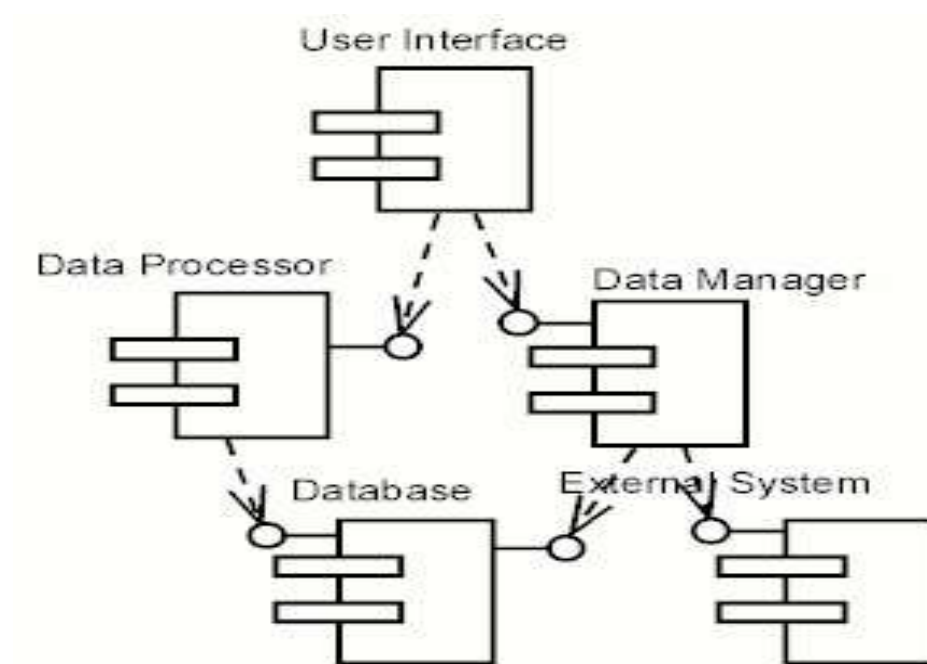
Decisions - (nodes in the transitions) which cause the thread to select one of multiple paths.

Swimlanes - (vertical lines the length of the diagram) which allow activities to be assigned to objects.

Synch States - horizontal or vertical solid lines which split or merge threads (transitions)

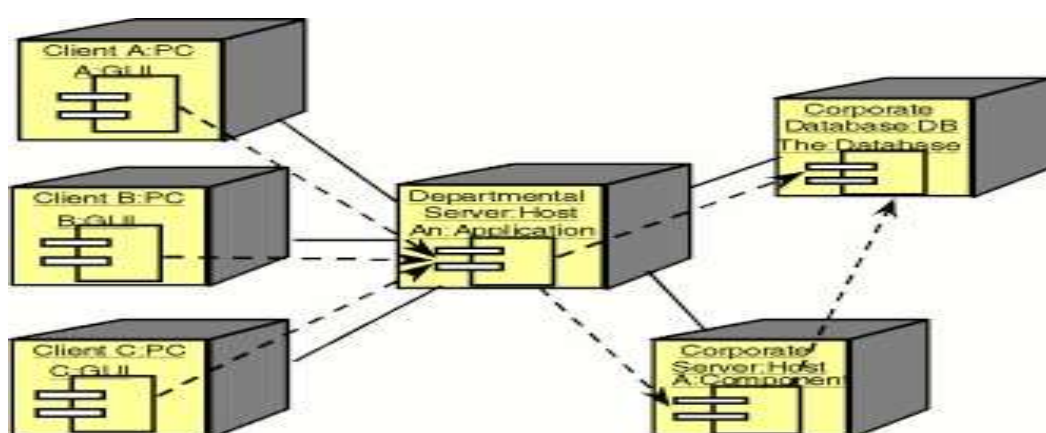
Component Diagram

Component Diagrams show the types of software components in the system, their interfaces and dependencies.



Deployment Diagram

Deployment diagrams show the computing nodes in the system, their communication links, the components that run on them and their dependencies.



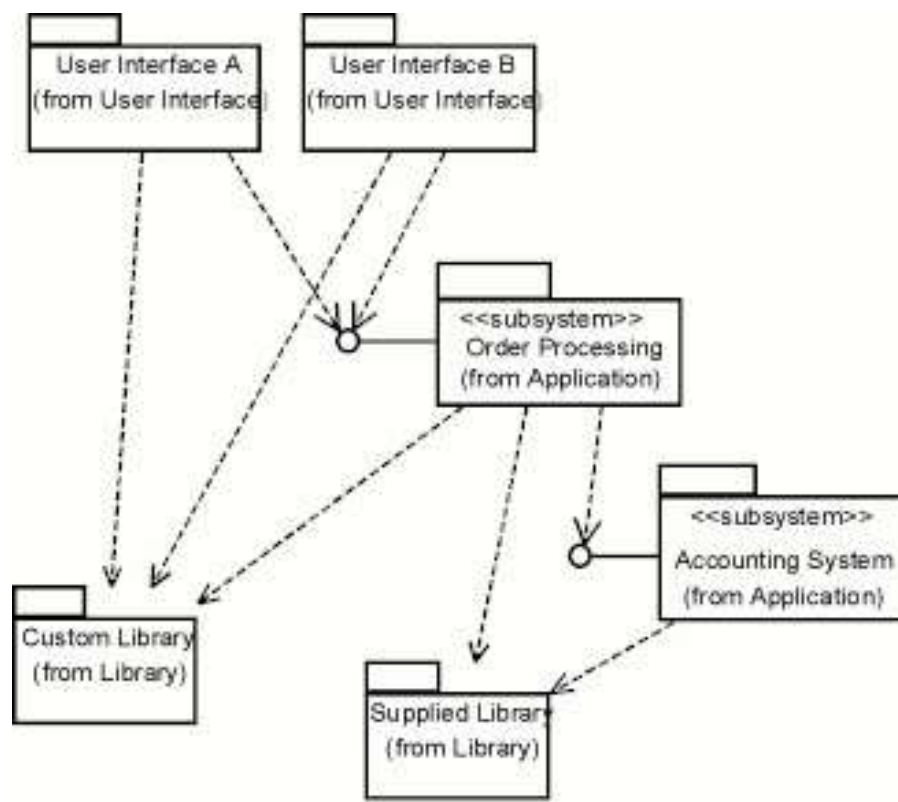
Modeling Architecture

Class diagrams can be used to model hi-level architecture with packages, interfaces and dependencies.

Packages are used to group together a set of model elements for various purposes. The results may show:

Subsystems - the design view of a software component or re-useable part of a component.





Libraries of re-useable elements, usually classes.



The hierarchic structure and layering of the system

Client-server relationships between components and other model elements

Logical dependencies of sub-systems and libraries on one another

Object Oriented Programming Using Java

What is Java?

- A high-level programming language developed by Sun Microsystems. Java was originally called OAK, and was designed for handheld devices and set-top boxes. Oak was unsuccessful so in 1995 Sun changed the name to Java and modified the language to take advantage of the burgeoning World Wide Web.
- Java is an object-oriented language similar to C++, but simplified to eliminate language features that cause common programming errors. Java source code files (files with a .java extension) are compiled into a format called bytecode (files with a .class extension), which can then be executed by a Java interpreter. Compiled Java code can run on most computers because Java interpreters and runtime environments, known as Java Virtual Machines (VMs), exist for most operating systems, including UNIX, the Macintosh OS, and Windows. Bytecode can also be converted directly into machine language instructions by a just-in-time compiler (JIT).
- Java is a general purpose programming language with a number of features that make the language well suited for use on the World Wide Web. Small Java applications are called Java applets and can be downloaded from a Web server and run on your computer by a Java-compatible Web browser, such as Netscape Navigator or Microsoft Internet Explorer.

Features of Java

1. Java is a Platform

Java (with a capital J) is a platform for application development. A platform is a loosely defined computer industry buzzword that typically means some combination of hardware and system software that will mostly run all the same software. For instance PowerMacs running Mac OS 9.2 would be one platform. DEC Alphas running Windows NT would be another.

Java solves the problem of platform-independence by using byte code. The Java compiler does not produce native executable code for a particular machine like a C compiler would. Instead it produces a special format called *byte code*. Java byte code written in hexadecimal, byte by byte, looks like this:

```
CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20 08
```

This looks a lot like machine language, but unlike machine language Java byte code is exactly the same on every platform. This byte code fragment means the same thing on a Solaris workstation as it does on a Macintosh PowerBook. Java programs that have been compiled into byte code still need an interpreter to execute them on any given platform. The interpreter reads the byte code and translates it into the native language of the host machine on the fly. The most common such interpreter is Sun's program `java` (with a little `j`). Since the byte code is completely platform independent, only the interpreter and a few native libraries need to be ported to get Java to run on a new computer or operating system. The rest of the runtime environment including the compiler and most of the class libraries are written in Java.

2. Java is Simple

Java was designed to make it much easier to write bug free code. Java has the functionality needed to implement its rich feature set. It does not add lots of syntactic sugar or unnecessary features. Despite its simplicity Java has considerably more functionality than C, primarily because of the large class library.

Because Java is simple, it is easy to read and write.

About half of the bugs in C and C++ programs are related to memory allocation and deallocation.

Therefore the second important addition Java makes to providing bug-free code is automatic memory allocation and deallocation. The C library memory allocation functions `malloc()` and `free()` are gone as are C++'s destructors.

Java is an excellent teaching language, and an excellent choice with which to learn programming. The language is small so it's easy to become fluent. The language is interpreted so the compile-run-link cycle is much shorter. The runtime environment provides automatic memory allocation and garbage collection so there's less for the programmer to think about. Java is object-oriented unlike Basic so the beginning programmer doesn't have to unlearn bad programming habits when moving into real world projects. Finally, it's very difficult (if not quite impossible) to write a Java program that will crash your system, something that you can't say about any other language.

3. Java is Object-Oriented

Object oriented programming is the catch phrase of computer programming in the 1990's. In object-oriented programs data is represented by objects. Objects have two sections, fields (instance variables) and methods. Fields tell you what an object is. Methods tell you what an object does. These fields and methods are closely tied to the object's real world characteristics and behavior. When a program is run messages are

passed back and forth between objects. When an object receives a message it responds accordingly as defined by its methods.

Object oriented programming is alleged to have a number of advantages including:

- Simpler, easier to read programs
- More efficient reuse of code
- Faster time to market
- More robust, error-free code

4. Java is Platform Independent

Java was designed to not only be cross-platform in source form like C, but also in compiled binary form. Since this is frankly impossible across processor architectures Java is compiled to an intermediate form called byte-code. A Java program never really executes natively on the host machine. Rather a special native program called the Java interpreter reads the byte code and executes the corresponding native machine instructions. Thus to port Java programs to a new platform all that is needed is to port the interpreter and some of the library routines. Even the compiler is written in Java. The byte codes are precisely defined, and remain the same on all platforms.

The second important part of making Java cross-platform is the elimination of undefined or architecture dependent constructs. Integers are always four bytes long, and floating point variables follow the IEEE 754 standard for computer arithmetic exactly. You don't have to worry that the meaning of an integer is going to change if you move from a Pentium to a PowerPC. In Java everything is guaranteed.

However the virtual machine itself and some parts of the class library must be written in native code. These are not always as easy or as quick to port as pure Java programs.

5. Java is Safe

Java was designed from the ground up to allow for secure execution of code across a network, even when the source of that code was untrusted and possibly malicious.

This required the elimination of many features of C and C++. Most notably there are no pointers in Java. Java programs cannot access arbitrary addresses in memory. All memory access is handled behind the scenes by the (presumably) trusted runtime environment. Furthermore Java has strong typing. Variables

must be declared, and variables do not change types when you aren't looking. Casts are strictly limited to casts between types that make sense. Thus you can cast an int to a long or a byte to a short but not a long to a boolean or an int to a String.

Java implements a robust exception handling mechanism to deal with both expected and unexpected errors. The worst that an applet can do to a host system is bring down the runtime environment. It cannot bring down the entire system.

Most importantly Java applets can be executed in an environment that prohibits them from introducing viruses, deleting or modifying files, or otherwise destroying data and crashing the host computer. A Java enabled web browser checks the byte codes of an applet to verify that it doesn't do anything nasty before it will run the applet.

6. Java is High Performance

Java byte codes can be compiled on the fly to code that rivals C++ in speed using a "just-in-time compiler." Several companies are also working on native-machine-architecture compilers for Java.

It is certainly possible to write large programs in Java. The HotJava browser, the Eclipse integrated development environment, the LimeWire file sharing application, the jEdit text editor, the JBoss application server, the Tomcat servlet container, the Xerces XML parser, the Xalan XSLT processor, and the javac compiler are large programs that are written entirely in Java.

7. Java is Multi-Threaded

Java is inherently multi-threaded. A single Java program can have many different threads executing independently and continuously. Three Java applets on the same page can run together with each getting equal time from the CPU with very little extra effort on the part of the programmer.

This makes Java very responsive to user input. It also helps to contribute to Java's robustness and provides a mechanism whereby the Java environment can ensure that a malicious applet doesn't steal all of the host's CPU cycles.

Unfortunately multithreading is so tightly integrated with Java, that it makes Java rather difficult to port to architectures like Windows 3.1 or the PowerMac that don't natively support preemptive multi-threading. There is a cost associated with multi-threading. Multi-threading is to Java what pointer arithmetic is to C, that is, a source of devilishly hard to find bugs. Nonetheless, in simple programs it's possible to leave multi-threading alone and normally be OK.

7. Java is dynamically linked

Java does not have an explicit link phase. Java source code is divided into .java files, roughly one per each class in your program. The compiler compiles these into .class files containing byte code. Each .java file generally produces exactly one .class file.

(There are a few exceptions non-public classes and inner classes).

The compiler searches the current directory and directories specified in the CLASSPATH environment variable to find other classes explicitly referenced by name in each source code file. If the file you're compiling depends on other, non-compiled files the compiler will try to find them and compile them as well. The compiler is quite smart, and can handle circular dependencies as well as methods that are used before they're declared. It also can determine whether a source code file has changed since the last time it was compiled.

More importantly, classes that were unknown to a program when it was compiled can still be loaded into it at runtime. For example, a web browser can load applets of differing classes that it's never seen before without recompilation.

Furthermore, Java .class files tend to be quite small, a few kilobytes at most. It is not necessary to link in large runtime libraries to produce a (non-native) executable. Instead the necessary classes are loaded from the user's CLASSPATH.

8. Java is Garbage Collected

You do not need to explicitly allocate or deallocate memory in Java. Memory is allocated as needed, both on the stack and the heap, and reclaimed by the *garbage collector* when it is no longer needed. There's no malloc(), free(), or destructor methods.

There are constructors and these do allocate memory on the heap, but this is transparent to the programmer. The exact algorithm used for garbage collection varies from one virtual machine to the next. The most common approach in modern VMs is generational garbage collection for short-lived objects, followed by mark and sweep for longer lived objects. I have never encountered a Java VM that used reference counting.

Applications and Applets

There are two types of programs implemented in java- applets and applications. The difference lies on how they are executed.

Applications: Application is a Java class that has a main() method. Generally, application is a stand-alone program, normally launched from the command line, and which has unrestricted access to the host system. Applications can execute independently.

Applet is a program which is run in the context of an applet viewer or web browser, and which has strictly limited access to the host system. Applets are used to provide interactive features to web applications that cannot be provided by HTML alone. They can capture mouse input and also have controls like buttons or check boxes. In response to the user action an applet can change the provided graphic content.

Requirements for Creating Java programs

- Text Editor: Notepad, Jcreator, emacs or vi. Personally I use BBEdit on the Mac and TextPad on Windows.
- Java Development Kit (JDK) : is a program development environment for writing Java applets and applications. It consists of a runtime environment that "sits on top" of the operating system layer as well as the tools and programming that developers need to compile, debug, and run applets and applications written in the Java language.
- Command line OS – for compiling and Execution environment.

Understanding structure of Java Application

Example 1

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

class Test

- **class**– Keyword class indicates the start of a class program
- **Test**– Class name ; Valid named to identify the class. Follow the rules for naming variables to name classes

- **public static void main (String args[])** – main method. It contains the following elements. Main () is the point where the execution of the program begin.
- **public**– indicate that main method can be accessed by any object.
- **static**– indicate that main method is a class method and that only one main can exist.
- **void** – indicate main method does not return a value.
- **String args[]** – argument to main method. It is array of type string. This array is mechanism through which runtime system passes information to your application.

Creating Your First Application

Your first application, `HelloWorldApp`, will simply display the greeting "Hello world!". To create this program, you will:

- **Create a source file**

A source file contains code, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.

- **Compile the source file into a .class file**

The Java programming language *compiler* (`javac`) takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as *bytecodes*.

- **Run the program**

The Java application *launcher tool* (`java`) uses the Java virtual machine to run your application.

Create a Source File

To create a source file, you have two options:

First, start your editor. You can launch the Notepad editor from the **Start** menu by selecting **Programs > Accessories > Notepad**. In a new document, type in the following code:

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Note: Type all code, commands, and file names exactly as shown. Both the compiler (javac) and launcher (java) are *case-sensitive* , so you must capitalize consistently.

HelloWorldApp is *not* the same as helloworldapp.

Save the code in a file with the name HelloWorldApp.java . To do this in Notepad, first choose the **File > Save As** menu item. Then, in the **Save As** dialog box:

1. Using the **Save in** combo box, specify the folder (directory) where you'll save your file. In this example, the directory is java on the C drive.
2. In the **File name** text field, type "HelloWorldApp.java", including the quotation marks.
3. From the **Save as type** combo box, choose **Text Documents (*.txt)** .
4. In the **Encoding** combo box, leave the encoding as ANSI.

When you're finished, the dialog box should look like this.

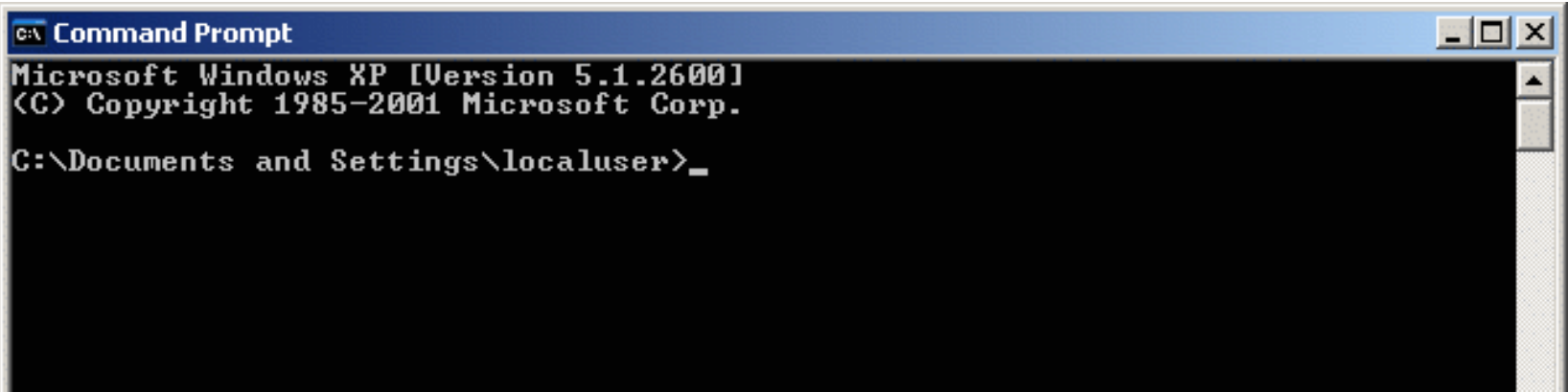


The Save As dialog just before you click **Save**.

Now click **Save**, and exit Notepad.

Compile the Source File into a .class File

Bring up a shell, or "command," window. You can do this from the **Start** menu by choosing **Command Prompt** (Windows XP), or by choosing **Run...** and then entering `cmd`. The shell window should look similar to the following figure.



A shell window.

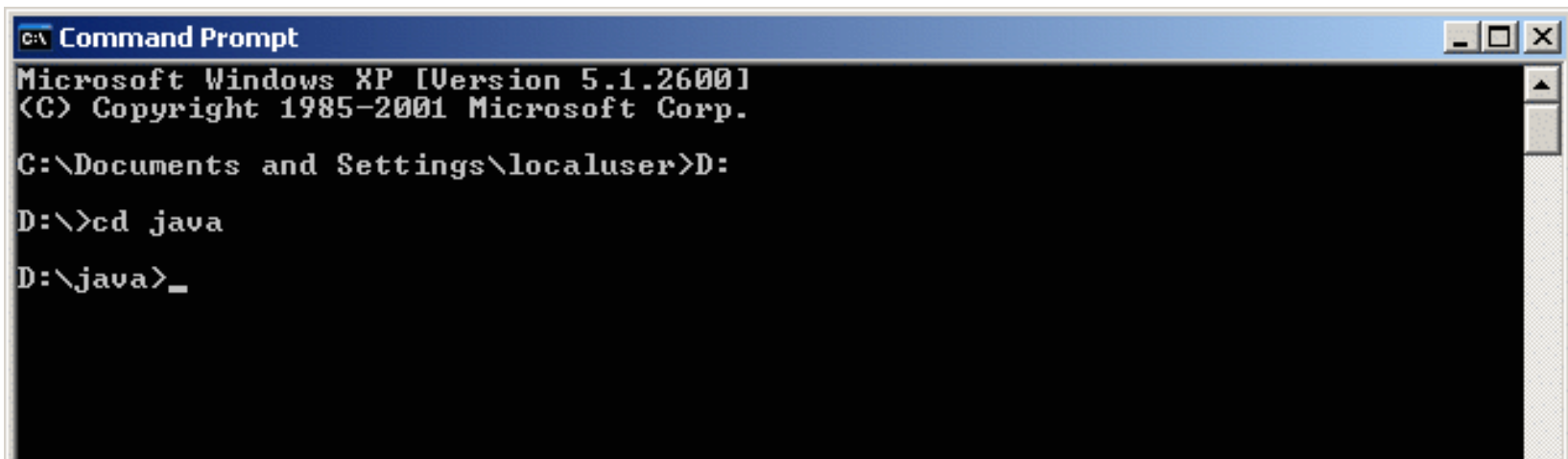
The prompt shows your *current directory*. When you bring up the prompt, your current directory is usually your home directory for Windows XP (as shown in the preceding figure).

To compile your source file, change your current directory to the directory where your file is located. For example, if your source directory is `java` on the `C` drive, type the following command at the prompt and press **Enter**:

```
cd C:\java
```

Now the prompt should change to `C:\java>`.

Note: To change to a directory on a different drive, you must type an extra command: the name of the drive. For example, to change to the `java` directory on the `D` drive, you must enter `D:`, as shown in the following figure.



Changing directory on an alternate drive.

If you enter `dir` at the prompt, you should see your source file, as the following figure shows.

```
C:\ Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\localuser>cd C:\java

C:\java>dir
Volume in drive C has no label.
Volume Serial Number is 242E-E457

Directory of C:\java

11/20/2005  08:43 PM    <DIR>          .
11/20/2005  08:43 PM    <DIR>          ..
11/20/2005  08:43 PM                284 HelloWorldApp.java
               1 File(s)                284 bytes
               2 Dir(s)  1,918,476,288 bytes free

C:\java>_
```

Directory listing showing the .java source file.

Now you are ready to compile. At the prompt, type the following command and press **Enter**.

```
javac HelloWorldApp.java
```

The compiler has generated a bytecode file, HelloWorldApp.class . At the prompt, type dir to see the new file that was generated, as shown in the following figure.

```
C:\ Command Prompt

C:\java>dir
Volume in drive C has no label.
Volume Serial Number is 242E-E457

Directory of C:\java

11/21/2005  12:36 PM    <DIR>          .
11/21/2005  12:36 PM    <DIR>          ..
11/21/2005  12:36 PM                432 HelloWorldApp.class
11/20/2005  08:43 PM                284 HelloWorldApp.java
               2 File(s)                716 bytes
               2 Dir(s)  1,479,315,456 bytes free

C:\java>
```

Directory listing, showing the generated .class file

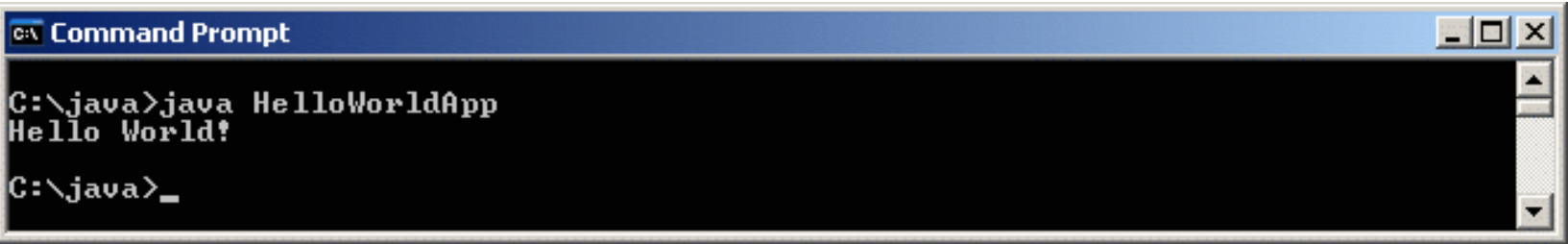
Now that you have a .class file, you can run your program.

Run the Program

In the same directory, enter the following command at the prompt:

java HelloWorldApp

The next figure shows what you should now see:



The program prints "Hello World!" to the screen.

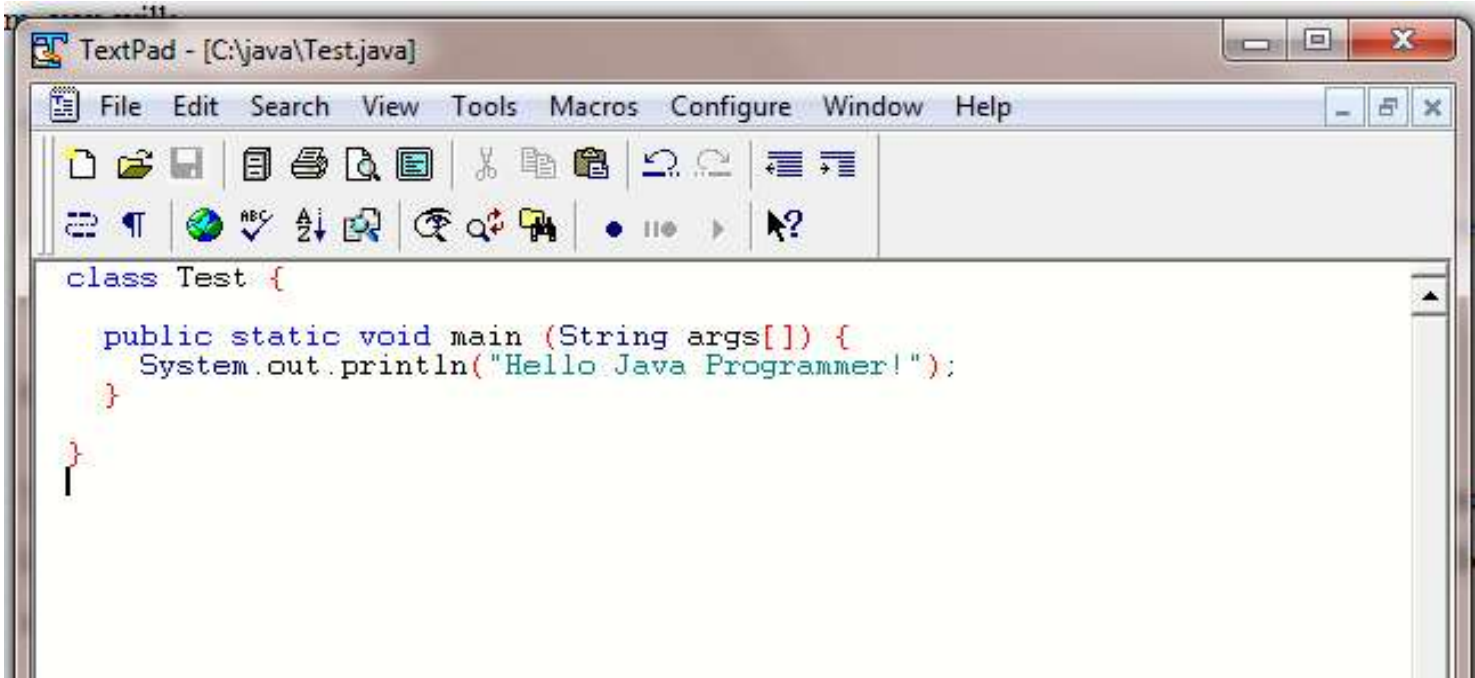
Congratulations! Your program works!

Compiling and Executing the Application in Windows Environment

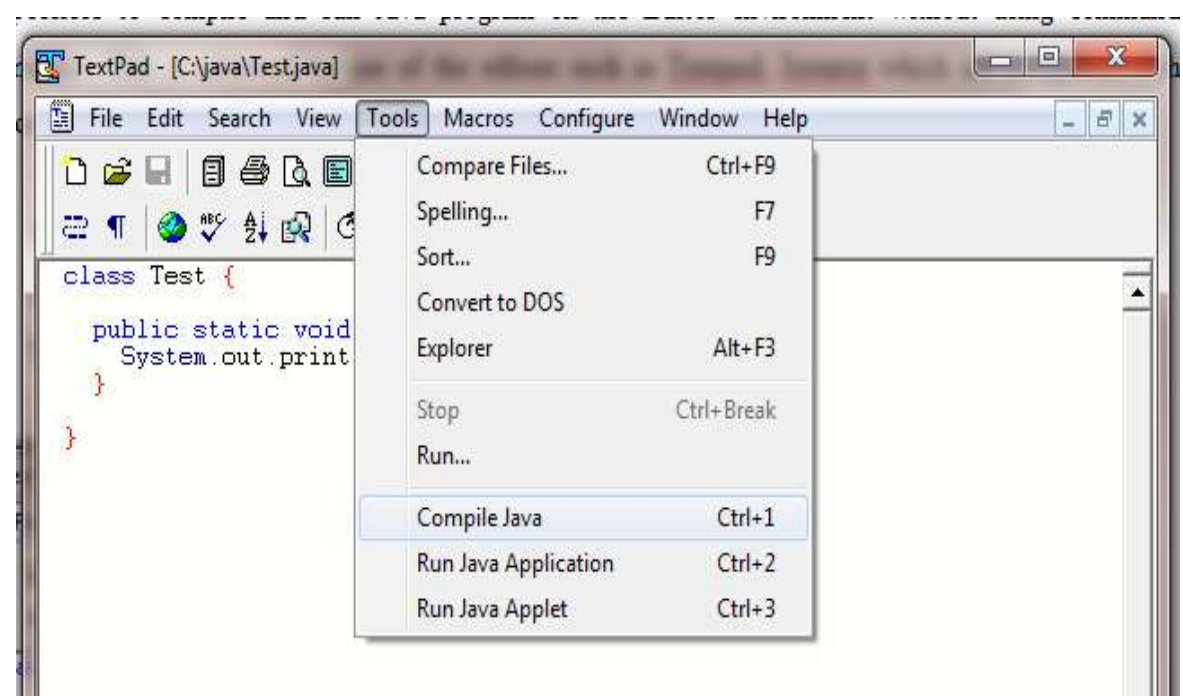
It is possible to compile and run Java program on the Editor environment without using command environment. This is achieved by use of the editors such as Textpad, Jcreator which are integrated with JDK components.

Steps

- 1. Write the source code
- 2. Save the source code e.g Test.java. (NB: Class name is always the file name).



- 3. To compile go to Tools > Compile Java.



4. To execute (run) the class file go to Tools > Run Java Application.

Output



Variables and Data types

Variables

Variable is a memory location for storing values which change during program execution.

The Java programming language defines the following kinds of variables:

- **Instance Variables (Non-Static Fields) :** objects store their individual states in "non-static fields", that is, fields declared without the static keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the currentSpeed of one bicycle is independent from the currentSpeed of another.
- **Class Variables (Static Fields)** A *class variable* is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as static since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.
- **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- **Parameters.** Recall that the signature for the main method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

Naming Variables

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- Variable names consists of letters and digits, and underscore only.
- First character must be a letter.
- Variable name must not be a keyword or reserved word.
- No space on the variable name – join with underscore or capitalization if word consists of two words. E.g
First_Name or FirstName.

Primitive Data Types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name,e.g

```
int gear = 1;
```

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1".

A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to `int`, the Java programming language supports seven other *primitive data types*. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

- **byte:** The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of `int` where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
- **short:** The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays, in situations where the memory savings actually matters.
- **int:** The `int` data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something

else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use `long` instead.

- **long:** The `long` data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you need a range of values wider than those provided by `int`.
- **float:** The `float` data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion. As with the recommendations for `byte` and `short`, use a `float` (instead of `double`) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency.
- **double:** The `double` data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
- **boolean:** The `boolean` data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- **char:** The `char` data type is a single 16-bit Unicode character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the **`java.lang.String`** class. Enclosing your character string within double quotes will automatically create a new `String` object; for example, `String s = "this is a string";`. `String` objects are *immutable*, which means that once created, their values cannot be changed. The `String` class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such.

Declaring variables

Involves specifying data type and variable name.

Syntax : datatype VarName;

Examples :

`int x;`

`double salary;`

`String Course;`

Default Values

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null , depending on the data type. Relying on such default values, however, is generally considered bad programming style.

The following chart summarizes the default values for the above data types.

Data Type Default Value (for fields)

byte 0

short 0

int 0

long 0L

float 0.0f

double 0.0d

char '\u0000'

String (or any object) null

boolean false

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

Literals

You may have noticed that the `new` keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise it is of type `int`. It is recommended that you use the upper case letter `L` because the lower case letter `l` is hard to distinguish from the digit `1`.

Values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Values of type `long` that exceed the range of `int` can be created from `long` literals. Integer literals can be expressed these number systems:

- Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day
- Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
- Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix `0x` indicates hexadecimal and `0b` indicates binary:

```
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
```

```
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating-Point Literals

A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise its type is `double` and it can optionally end with the letter `D` or `d`.

The floating point types (`float` and `double`) can also be expressed using `E` or `e` (for scientific notation), `F` or `f` (32-bit float literal) and `D` or `d` (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but
// in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

Character and String Literals

Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as `\u0108` (capital C with circumflex), or `"S\u00ED Se\u00F1or"` (Sí Señor in Spanish). Always use 'single quotes' for `char` literals and "double quotes" for `String` literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in `char` or `String` literals.

The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

There's also a special `null` literal that can be used as a value for any reference type. `null` may be assigned to any variable, except variables of primitive types. There's little you can do with a `null` value beyond testing for its presence. Therefore, `null` is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending ".class"; for example, `String.class`. This refers to the object (of type `Class`) that represents the type itself.

Initializing variables

This involves allocating some initial values to variables. Use the assignment operator (`=`).

Examples

```
int x=3;
```

```
double height=7.5;
```

```
String fname="Kennedy";
```

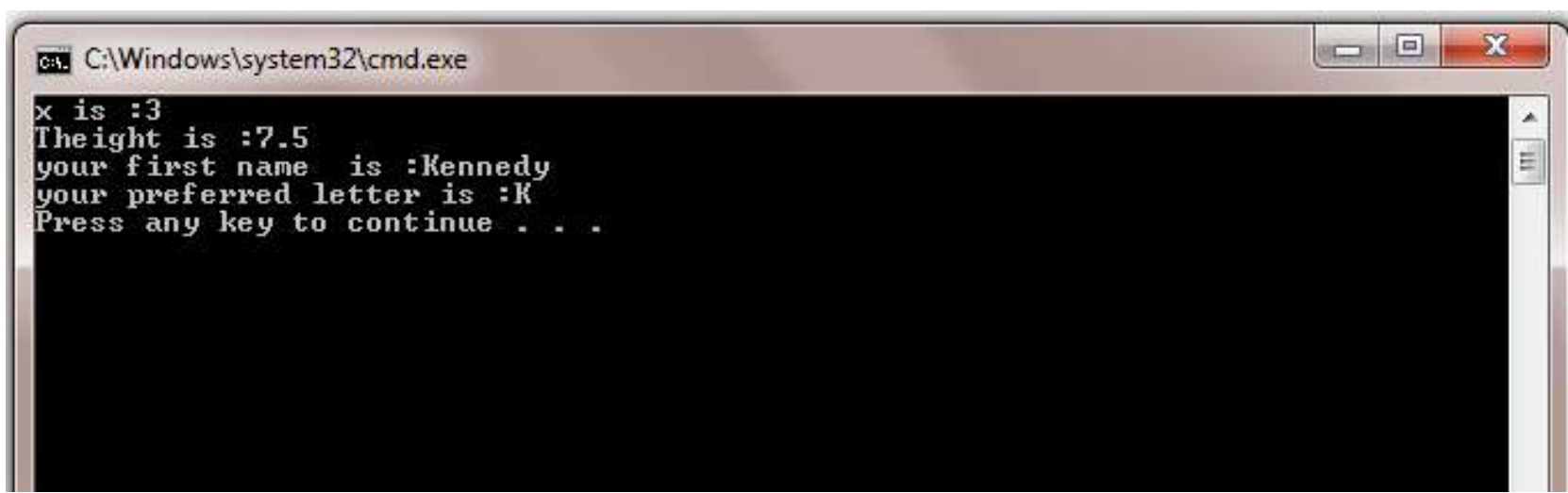
```
char letter = 'K';
```

Printing out variables variables

Use operator (+) to separate string constant with variable names

Example

```
class VarTest{
public static void main (String args[])
{
//declare and initialize variables
int x=3;
double height=7.5;
String fname="Kennedy";
char letter = 'K';
//display variables values
System.out.println("x is :"+x);
System.out.println("Theight is :"+height);
System.out.println("your first name  is :"+fname);
System.out.println("your preferred letter is :"+letter);
}
}
```

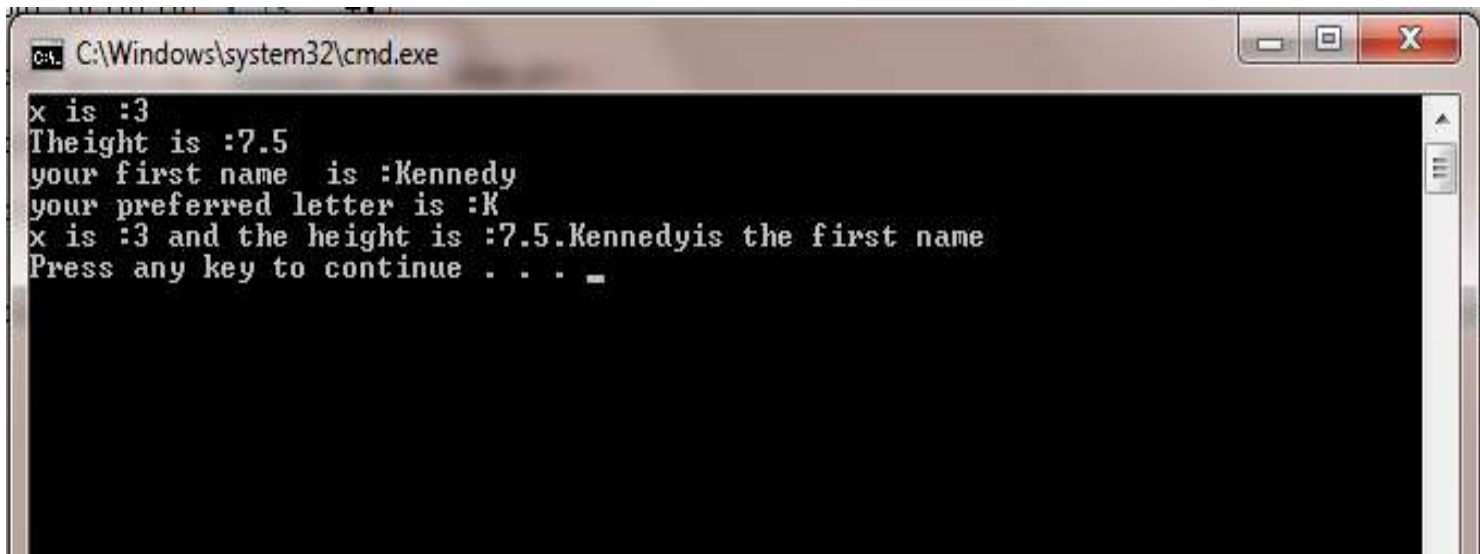
```
C:\Windows\system32\cmd.exe
x is :3
Theight is :7.5
your first name is :Kennedy
your preferred letter is :K
Press any key to continue . . .
```

Using multiple variables in the same statement

Example

```
class VarTest{
public static void main (String args[]) {
int x=3;
double height=7.5;
String fname="Kennedy";
char letter = 'K';
System.out.println("x is :"+x);
System.out.println("Theight is :"+height);
System.out.println("your first name is :"+fname);
System.out.println("your preferred letter is :"+letter);

//after using variables in same statement.
System.out.println("x is :"+x+ " and the height is :"+ height+"."+fname
+ "is the first name");
}
}
```

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The text displayed is the output of a Java program: "x is :3", "Theight is :7.5", "your first name is :Kennedy", "your preferred letter is :K", "x is :3 and the height is :7.5.Kennedyis the first name", and "Press any key to continue . . . _".

```
C:\Windows\system32\cmd.exe
x is :3
Theight is :7.5
your first name is :Kennedy
your preferred letter is :K
x is :3 and the height is :7.5.Kennedyis the first name
Press any key to continue . . . _
```

Constants

A *constant variable* or *constant* is a variable whose value never changes (which may seem strange given the meaning of the word *variable*). Constants are useful for defining shared values for all the methods of an object-for giving meaningful names to objectwide values that will never change. In Java, you can create constants only for instance or class variables, not for local variables.

New Term

A *constant* is a variable whose value never changes.

To declare a constant, use the `final` keyword before the variable declaration and include an initial value for that variable:

```
final float pi = 3.141592;
final boolean debug = false;
final int maxsize = 40000;
```

Technical Note

The only way to define constants in Java is by using the `final` keyword. Neither the C and C++ constructs for `#define` nor `const` are available in Java, although the `const` keyword is reserved to prevent you from accidentally using it.

Constants can be useful for naming various states of an object and then testing for those states. For example, suppose you have a test label that can be aligned left, right, or center. You can define those values as constant integers:

```
final int LEFT = 0;
final int RIGHT = 1;
final int CENTER = 2;
```

The variable alignment is then also declared as an `int` :

```
int alignment;
```

Then, later in the body of a method definition, you can either set the alignment:

```
this.alignment = CENTER;
```

or test for a given alignment:

```
switch (this.alignment) {
case LEFT: // deal with left alignment
...
break;
case RIGHT: // deal with right alignment
...
break;
case CENTER: // deal with center alignment
...
break;
}
```

Using Keyboard Input

Java provides different methods for capturing user input through the keyboard. They include

- showInputDialog function
- readLine Function
- scanner function.

Using showInputDialog function

- This function is supported by javax.swing package.
- The class used is JOptionPane of swing package.
- Therefore the package needs to be imported on the header as follows:

```
import javax.JOptionPane;
```

Example

```
import javax.swing.JOptionPane;

public class InputBox
{
    public static void main(String args[])
    {
        String fnum, snum;
        int num1, num2, sum;
        fnum = JOptionPane.showInputDialog("Enter the first Number");
        snum=JOptionPane.showInputDialog("Enter the second Number");
        num1=Integer.parseInt(fnum);
        num2=Integer.parseInt(snum);
        sum=num1+num2;
        JOptionPane.showMessageDialog(null, "the Sum is :" + sum,
        "Addition Results", JOptionPane.PLAIN_MESSAGE);
        System.exit(0);
    }
}
```

Explanation

- import javax.swing.JOptionPane – include the class needed to support the function showInputDialog.
- String fnum, snum; -declare the variables of type string to capture using keyboard.

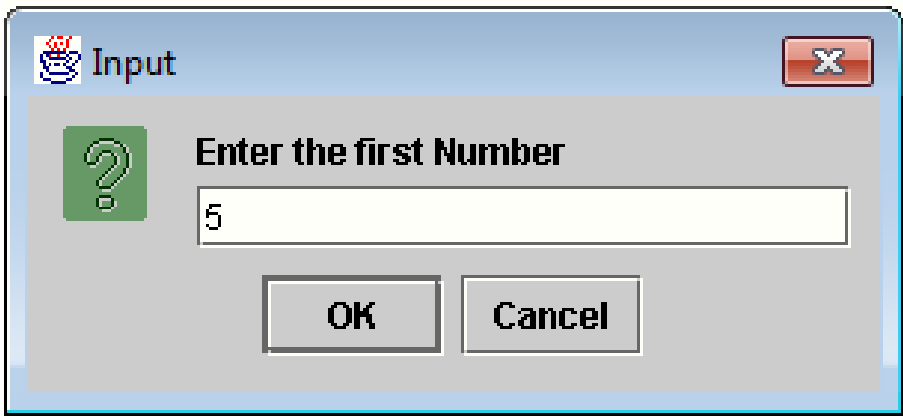
NB: all keyboard values are captured as string in Java then vonverted to other types using conversion functions.

- int num1, num2, sum; - declare equivalent integer variables for fnum AND snum. They will hold the converted values.
- Sum – will hold the results of the addition of num1 and num2.
- fnum = JOptionPane.showInputDialog("Enter the first Number");
- snum=JOptionPane.showInputDialog("Enter the second Number");
 - Creates input box to allow user to capture the value
- num1=Integer.parseInt(fnum); - Convert string value fnum to integer using function parseInt()

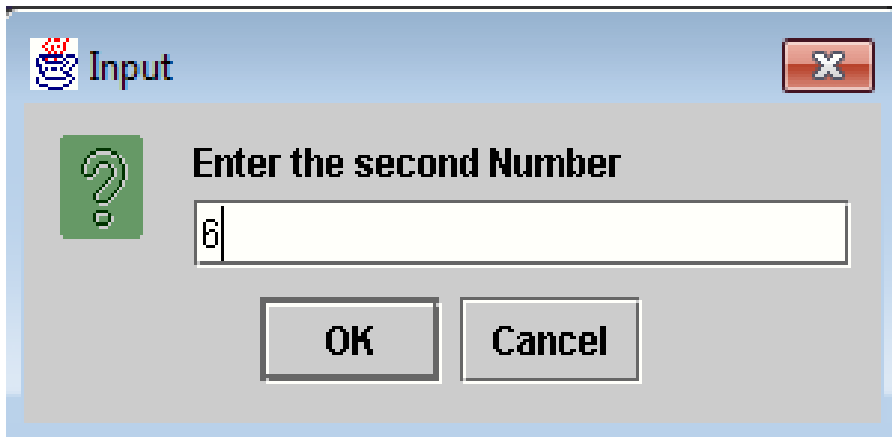
JOptionPane.showMessageDialog(null, "the Sum is :" + sum, "Addition Results", JOptionPane.PLAIN_MESSAGE);

- Creates message box to display the results. This uses showMessageDialog

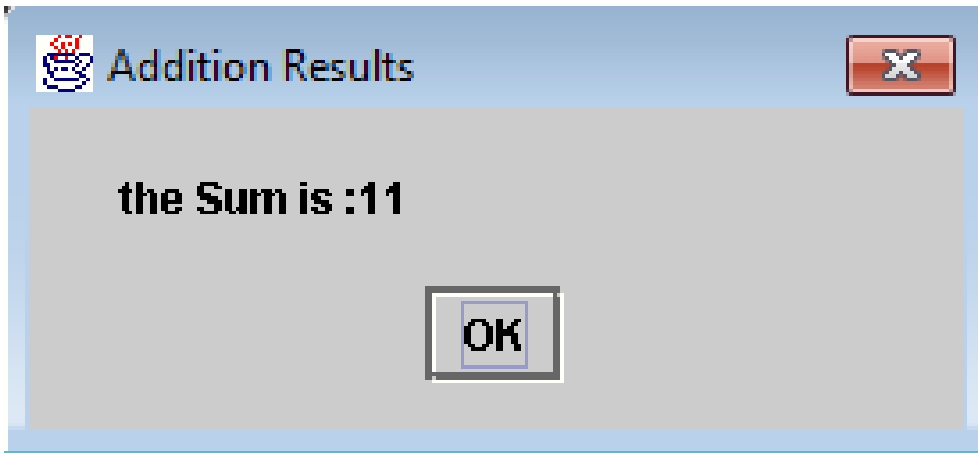
Input the value of fnum



Input the value of snum



Display the sum using message box



Using readLine Function

The function is supported by java.io package therefore it needs to be imported first as follows:

```
import java.io.*;
```

Example:

```
import java.io.*;

class inputTest{

public static void main(String args[])throws IOException
{

BufferedReader input=new BufferedReader (new InputStreamReader(System.in));

//declare variables

String strnum1;

String strnum2;

int num1;

int num2;

int sum=0;

System.out.println("Enter first Number:");

strnum1=input.readLine();

num1=Integer.parseInt(strnum1);

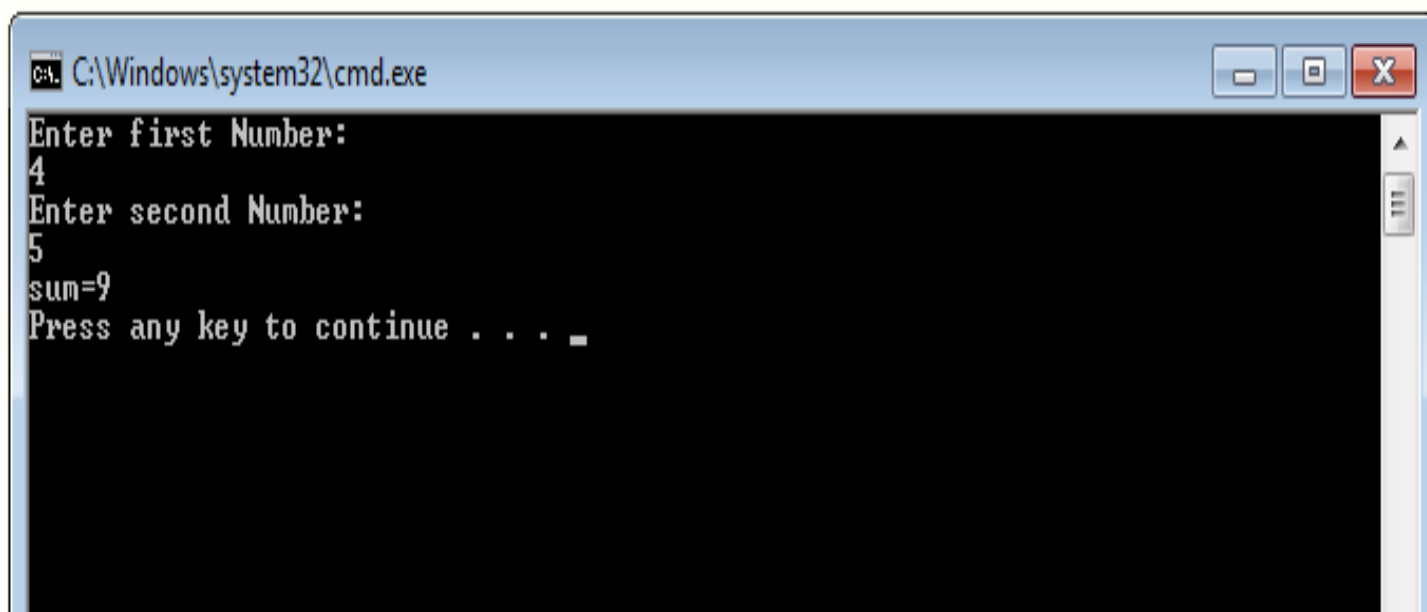

System.out.println("Enter second Number:");

strnum2=input.readLine();

num2=Integer.parseInt(strnum2);

sum=num1+num2;
```

```
System.out.println("sum="+sum);
}
}
```



Explanation

- `import java.io.*;` - include the required package.
 - `public static void main(String args[])throws IOException` – modified
main method to provide mechanism for handling input/output errors.
 - `BufferedReader input` – create object input of type `BufferedReader`.
`BufferedReader` is class of package `java.io`. it is required for `InputStreamReader` or `OutputStreamReader` streams.
 - `new BufferedReader (new InputStreamReader(System.in));` - Initialize the input object.
- NB: the input object is simply a variable name. any valid variable name can be used instead of input.
- `strnum1=input.readLine();` - use the object input to capture string value and assign it to variable `strnum1`.
 - `num1=Integer.parseInt(strnum1);` - covert the captured value into integer and assign the value to variable `num1`.

Exercise;

- (a) Write a program to let the user capture personal details : names, address, telephone, age, city, then display them. Use both readline and showInputDialog functions.
- (b) Write a program to compute percentage discount given the marked price and selling price. The user inputs marked price and selling price using keyboard.

Control Structures

Control structures determine the flow of program execution. By default statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control structures* (*Control flow statements*), however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the types of control structures:

- Sequence
- Selection/decision-making statements (`if` `switch`),
- Iterations (looping statements) - `for` , `while` , `do-while`

Selection/Decision Structure

The `if` Statement

The `if` statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to `true` . For example, the `Bicycle` class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the `applyBrakes` method could be as follows:

Example

```
void applyBrakes() {  
  // the "if" clause: bicycle  
  // must be moving  
  if (isMoving){  
    // the "then" clause: decrease  
    // current speed  
    currentSpeed--;  
  }  
}
```

If this test evaluates to `false` (meaning that the bicycle is not in motion), control jumps to the end of the `if` statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes() {  
    // same as above, but  
    // without braces  
    if (isMoving)  
        currentSpeed--;  
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

Example 2

```
int age  
if (age>=18)  
{  
    System.out.println("adult");  
}
```

The if-else Statement

The `if else` statement provides a secondary path of execution when an "if" clause evaluates to `false`.

You could use an `if else` statement in the `applyBrakes` method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes() {  
    if (isMoving) {  
        currentSpeed--;  
    } else {  
        System.err.println("The bicycle has " + "already stopped!");  
    }  
}
```

Example

```
int age
if (age>=18)
{
System.out.println(“adult”);
}
else
System.out.println(“child”);
```

If- else if – else statement

This is applicable where there are more than two statements to choose from.

Syntax

```
If (condition1)
Statement1;
else if(condition2)
Statement2;
else if(condition3)
Statement3;
.
.
.
.
else
Statement n;
```

The following program assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseIFDemo {
public static void main(String[] args) {

int testscore = 76;
char grade;
```

```
if (testscore >= 90) {  
    grade = 'A';  
} else if (testscore >= 80) {  
    grade = 'B';  
} else if (testscore >= 70) {  
    grade = 'C';  
} else if (testscore >= 60) {  
    grade = 'D';  
} else {  
    grade = 'F';  
}  
System.out.println("Grade = " + grade);  
}  
}
```

The output from the program is:

Grade = C

You may have noticed that the value of `testscore` can satisfy more than one expression in the compound statement: `76 >= 70` and `76 >= 60`. However, once a condition is satisfied, the appropriate statements are executed (`grade = 'C';`) and the remaining conditions are not evaluated.

Switch Statement

Another way to control the flow of your program is with a switch statement. A switch statement gives you the option to test for a range of values for your variables. They can be used instead of long, complex **if ... else if** statements. The structure of the switch statement is this:

```
switch ( variable_to_test ) {  
case value1:  
    code_here;  
    break;  
case value2:  
    code_here;  
    break;  
default:  
    values_not_caught_above;
```

}

So you start with the word **switch**, followed by a pair of round brackets. The variable you want to check goes between the round brackets of switch. You then have a pair of curly brackets. The other parts of the switch statement all go between the two curly brackets. For every value that you want to check, you need the word **case**. You then have the value you want to check for:

casevalue:

e.g

case 1:

case 2:

case 3:

case 'a':

case 'b':

case "Kenya":

case "Uganda"

After case value comes a colon. You then put what you want to happen if the value matches. This is your code that you want executed. The keyword **break** is needed to break out of each case of the switch statement.

The default value at the end is optional. It can be included if there are other values that can be held in your variable but that you haven't checked for elsewhere in the switch statement.

Example:

```

public static void main(String[] args) {

    int user = 18;

    switch ( user ) {
        case 18:
            System.out.println("You're 18");
            break;
        case 19:
            System.out.println("You're 19");
            break;
        case 20:
            System.out.println("You're 20");
            break;
        default:
            System.out.println("You're not 18, 19 or 20");
    }
}

```

The first thing the code does is to set a value to test for. Again, we've set up an integer variable and called it **user**. We've set the value to 18. The switch statement will check the user variable and see what's in it. It will then go through each of the case statements in turn. When it finds one that matches, it will stop and execute the code for that case. It will then break out of the switch statement.

Try the programme out. Enter various values for the user variable and see what happens.

Sadly, you can't test for a range of values after case, just the one value. So you can't do this:

case (user <= 18):

But you can do this:

case 1: case 2: case 3: case 4:

So the above line tests for a range of values, from 1 to 4. But you have to "spell out" each value. (Notice where all the case and colons are.)

To end this section on conditional logic, try these exercises.

Example

```
class switchTest2{
public static void main(String arg[])
{
int num;
num=3;
switch(num)
{
case 1:
System.out.println("Number=1 ");
break;

case 2:
System.out.println("Number=2");
break;
case 3:
System.out.println("Number=3");
break;

case 4:
System.out.println("Number=4");
break;
case 5:
System.out.println("Number=5");
break;
default:
System.out.println("No match for your number");
}
}
}
```

Example : shows how a statement can have multiple case labels. The code example calculates the number of days in a particular month:

```
class SwitchDemo2 {
public static void main(String[] args) {

int month = 2;
int year = 2000;
int numDays = 0;

switch (month) {
case 1: case 3: case 5:
case 7: case 8: case 10:
case 12:
numDays = 31;
```

```

break;
case 4: case 6:
case 9: case 11:
numDays = 30;
break;
case 2:
if (((year % 4 == 0) &&
!(year % 100 == 0))
|| (year % 400 == 0))
numDays = 29;
else
numDays = 28;
break;
default:
System.out.println("Invalid month.");
break;
}
System.out.println("Number of Days = "
+ numDays);
}
}

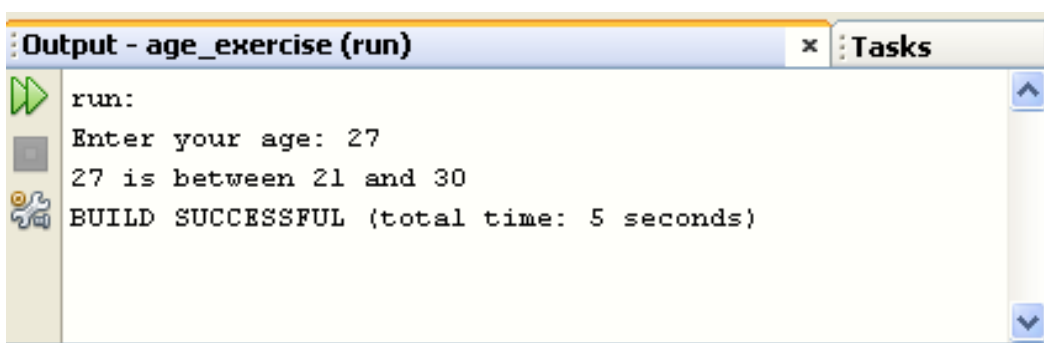
```

Exercise

Write a programme that accepts user input from the console. The programme should take a number and then test for the following age ranges: 0 to 10, 11 to 20, 21 to 30, 30 and over. Display a message in the Output window in the following format:

user_age + " is between 21 and 30"

So if the user enters 27 as the age, the Output window should be this:



If the user is 30 or over, you can just display the following message:

"Your are 30 or over"

Help for this exercise

To get string values from the user, you did this:

```
String age = user_input.next( );
```

But the **next()** method is used for strings. The age you are getting from the user has to be an integer, so you can't use **next()**. There is, however, a similar method you can use: **nextInt()**.

Exercise

If you want to check if one String is the same as another, you can use a Method called **equals**.

```
String user_name = "Bill";
```

```
if ( user_name.equals( "Bill" )) {  
//DO SOMETHING HERE  
}
```

In the code above, we've set up a String variable and called it `user_name`. We've then assigned a value of "Bill" to it. In between the round brackets of IF we have the variable name again, followed by a dot. After the dot comes the word "equals". In between another pair of round brackets you type the string you're trying to test for.

NOTE: When checking if one string is the same as another, they have to match exactly. So "Bill" is different from "bill". The first has an uppercase letter "B" and the second has a lowercase "b".

For this exercise, write a program that asks a user to choose between four colours: black, white, red, or blue. Use IF ... ELSE IF statements to display one of the following messages, depending on which colour was chosen:

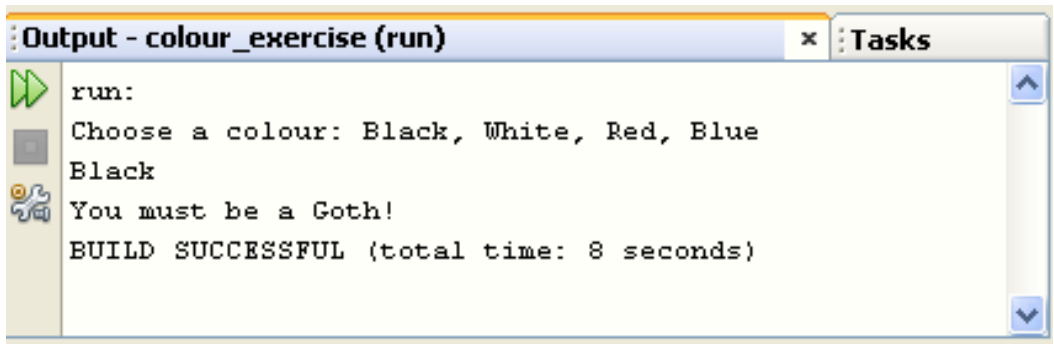
BLACK"You must be a Goth!"

WHITE"You are a very pure person"

RED"You are fun and outgoing"

BLUE"You're not a Chelsea fan, are you?"

When your programme ends, the Output window should look like something like this:



Loops

The purpose of loop statements is to *repeat Java statements* a given number of times until certain conditions occur. A programming loop is one that forces the program to go back up again. If it is forced back up again you can execute lines of code repeatedly. There are three kinds of loop statements in Java.

While loop

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression)
{
    statement(s)
}
```

NB: In while loop, the test is done at the start. i.e. test then execute.

The while statement evaluates *expression*, which must return a *boolean* value. If the expression evaluates to *true*, the while statement executes the *statement(s)* in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to *false*. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.print(count + " ");
            count++;
        }
    }
}
```

```
System.out.println("Count is: " + count);
count++;
}
}
}
```

You can implement an infinite loop using the `while` statement as follows:

```
while (true){
// your code goes here
}
```

Do while

The Java programming language also provides a `do-while` statement, which can be expressed as follows:

```
do {
statement(s)
} while (expression);
```

The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once, as shown in the following `DoWhileDemo` program:

```
class DoWhileDemo {
public static void main(String[] args){
int count = 1;
do {
System.out.println("Count is: " + count);
count++;
} while (count <= 11);
}
}
```

Example :

```
class DoWhileTest{

public static void main(String arg[])

{

int count=1;
```

```
do
{
System.out.println("Hello number :"+count);

count++;

} while(count<=5);

System.out.println("End of Hello!!");

}

}
```

For Loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

```
for(initialization; Boolean_expression; update)
{
//Statements
}
```

Here is the flow of control in a for loop:

1. The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
4. The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

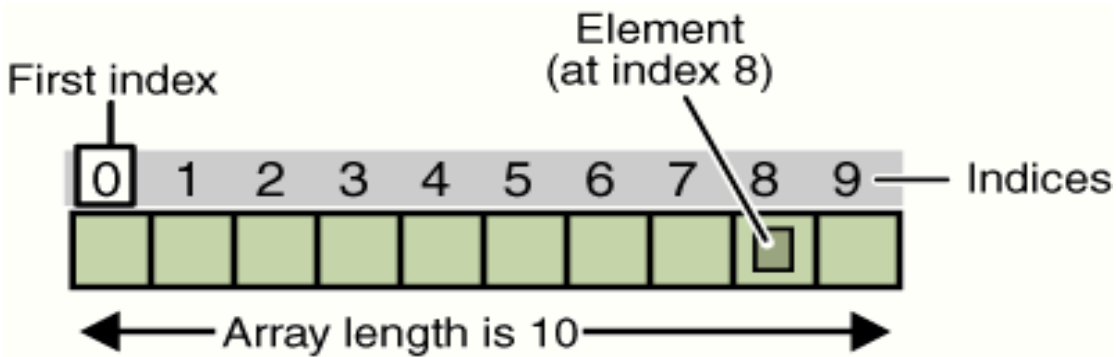
```
public class Test {  
    public static void main(String args[]){  
  
        for(int x = 10; x < 20; x = x+1)  
        {  
            System.out.print("value of x : " + x );  
            System.out.print("\n");  
        }  
    }  
}
```

This would produce following result:

```
value of x : 10  
value of x : 11  
value of x : 12  
value of x : 13  
value of x : 14  
value of x : 15  
value of x : 16  
value of x : 17  
value of x : 18  
value of x : 19
```

Arrays

- An *array* is a container object that holds a fixed number of values of a single type. The array has some number of slots, each of which holds an individual item.
- Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.



- The length of an array is established when the array is created. After creation, its length is fixed.

Characteristics of Array

- Elements are homogeneous –must same data type, any basic data type.
- Elements have random accessibility by use of index.
- It is fixed data structure – size is determined at compile time.
- Elements have sequential ordering – ‘ followed by’ attribute.

Creating Array

To create an array in Java, you use three steps:

1. Declare a variable to hold the array.
2. Create a new array object and assign it to the array variable.
3. Store things in that array.

Declaring Array Variables

The first step in creating an array is creating a variable that will hold the array, just as you would any other variable. Array variables indicate the type of object the array will hold (just as they do for any variable) and the name of the array, followed by empty brackets (`[]`). The following are all typical array variable declarations:

Syntax `data type ArrayName [];` or `data type [] ArrayName;`

```
String difficultWords[];
```

```
Point hits[];
```

```
int temps[];
```

An alternate method of defining an array variable is to put the brackets after the type instead of after the variable. They are equivalent, but this latter form is often much more readable. So, for example, these three declarations could be written like this:

```
String[] difficultWords;
```

```
Point[] hits;
```

```
int[] temps;
```

Creating Array Objects

The second step is to create an array object and assign it to that variable. There are two ways to do this:

- Using `new`
- Directly initializing the contents of that array

The first way is to use the `new` operator to create a new instance of an array:

```
String[] names = new String[10];
```

That line creates a new array of `String` s with 10 slots (sometimes called elements). When you create a new array object using `new` , you must indicate how many slots that array will hold. This line does not put actual `String` objects in the slots-you'll have to do that later.

Array objects can contain primitive types such as integers or booleans, just as they can contain objects:

```
int[] temps = new int[99];
```

When you create an array object using `new` , all its slots are initialized for you (`0` for numeric arrays, `false` for boolean, `\0` for character arrays, and `null` for objects). You can then assign actual values or objects to the slots in that array. You can also create an array and initialize its contents at the same time.

Instead of using `new` to create the new array object, enclose the elements of the array inside braces, separated by commas:

```
String[] chiles = { "jalapeno", "anaheim", "serrano",  
"habanero", "thai" };
```

Each of the elements inside the braces must be of the same type and must be the same type as the variable that holds that array (the Java compiler will complain if they're not). An array the size of the number of

elements you've included will be automatically created for you. This example creates an array of String objects named *chile* that contains five elements.

Another Example

```
int numbers []; //declare the array
```

```
int [] numbers = new int[10]; //create array object and array size
```

```
numbers[]=1; // store value on the array.
```

```
numbers[]=2;
```

```
numbers[]=3;
```

The three steps can be combined into one as follows

```
int numbers []={1,2,3,4,5,7,3,8,9,6};
```

Accessing Array Elements

- Once you have an array with initial values, you can test and change the values in each slot of that array.
- To get at a value stored within an array, use the array subscript expression (`myArray[subscript]`): - the **index**.

```
myArray[subscript];
```

- The `myArray` part of this expression is a variable holding an array object, although it can also be an expression that results in an array.
- The `subscript` part of the expression, inside the brackets, specifies the number of the slot within the array to access.
- Array subscripts start with `0`, as they do in C and C++. So, an array with 10 elements has 10 array slots accessed using subscript `0` to `9`.

E.g `numbers [0];` refers to the first element, 1.

Changing Array Elements

To assign an element value to a particular array slot, merely put an assignment statement after the array access expression:

```
myarray[1] = 15;
```



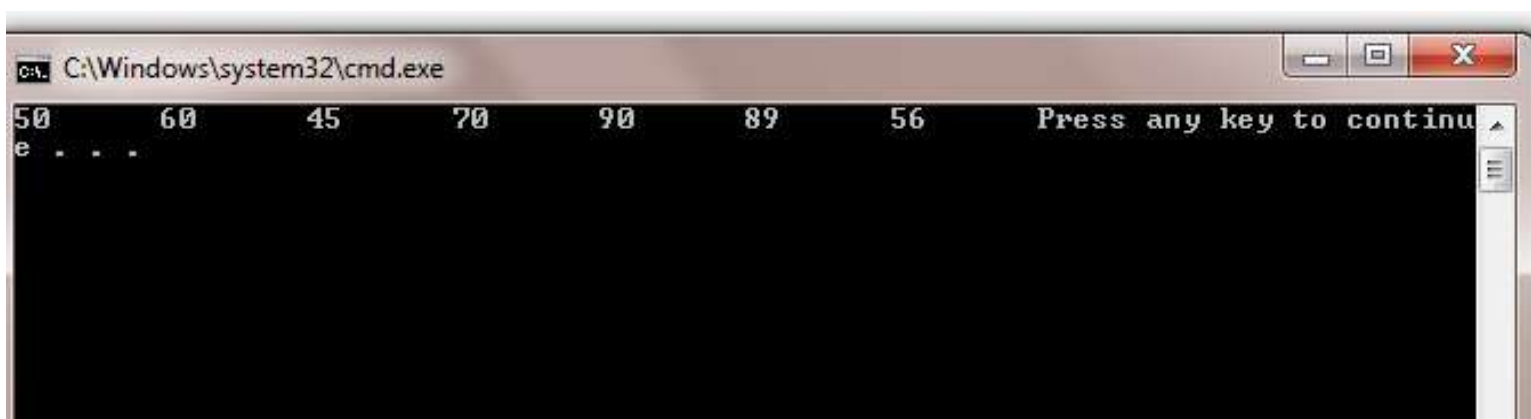
```
sentence[0] = "The";  
sentence[10] = sentence[0];
```

Processing array

- This involves printing out the values, sorting, searching, and computations (sum, average, maximum, minimum).
- This is done by use of *for loop*.

Example1: Print out the array elements.

```
class ArrayTest{  
  
    public static void main(String arg[])  
    {  
        int marks[]={ 50,60,45,70,90,89,56};  
  
        int index;  
  
        for(index=0;index<marks.length;index++)  
        {  
            System.out.print(marks[index]+"\\t");  
        }  
    }  
}
```



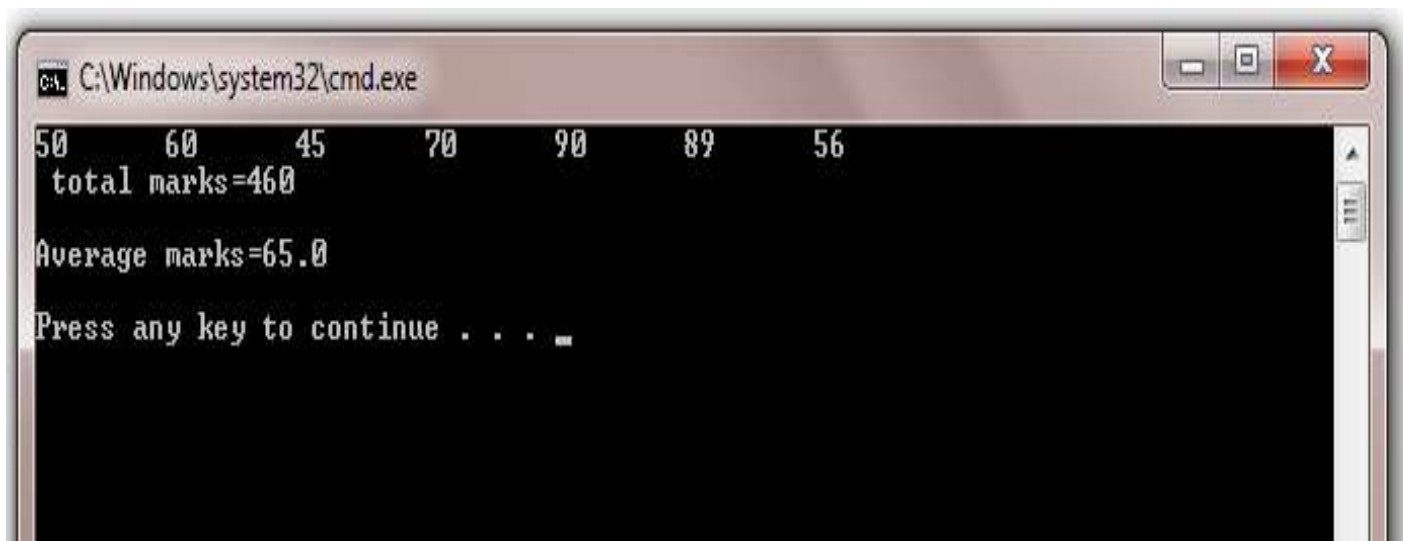
Example2: Print out the array elements, compute sum and average

```
class ArrayTest{

public static void main(String arg[])
{
int marks[]={ 50,60,45,70,90,89,56};

int index;
int sum=0;
double avg=0.0;

for(index=0;index<marks.length;index++)
{
System.out.print(marks[index]+"\\t");
sum=sum+marks[index];
}
avg=sum/marks.length;
System.out.println("\\n total marks="+sum);
System.out.println("");
System.out.println("Average marks="+avg);
System.out.println("");
}
}
```



```
C:\Windows\system32\cmd.exe
50    60    45    70    90    89    56
total marks=460
Average marks=65.0
Press any key to continue . . . .
```

Example 3 : Array program that allow user input of the elements, compute sum and average

```
import java.io.*;
class ArrayInput{
public static void main(String args[])throws IOException
{
BufferedReader input=new BufferedReader (new InputStreamReader(System.in));

double num[]; //array declaration
int size; //variable to specify the size of the array
System.out.println("Enter the size of the array:");

size=Integer.parseInt(input.readLine()); //input and parse array size
num=new double[size]; //allocate array size

double sum=0.0; double avg=0.0; //declare and initialize sum and average

for(int ctr=0;ctr<num.length;ctr++) //loop to track the num of value
entered
{
System.out.println("Enter the values:");
num[ctr]=Double.parseDouble(input.readLine());

sum=sum+num[ctr];

}

avg=sum/size;

System.out.println("-----");

System.out.println("sum="+sum);
System.out.println("Average="+avg);
System.out.println("-----");
}
}
```

Array Dimensions

- Java, as with most languages, supports multi-dimensional arrays - 1-dimensional, 2-dimensional, 3-dimensional.

- This discusses 2-dimensional arrays, but the same principles apply to higher dimensions.

2-dimensional arrays

2-dimensional arrays are usually represented in a row-column approach on paper, and the terms "rows" and "columns" are used in computing.

```
int numbers[ ][ ]; or int [ ][ ] numbers; // declare 2- dimensional array.
```

```
int numbers [ ][ ] = new int [3] [4]; // define array size.
```

This is an array of 3 rows and 4 columns.

Maximum size of array is given by $3 \times 4 = 12$.

Declaration

Declare a 2-dimensional array as follows:

Code: Java

```
int[][] a2;           // Declares, but doesn't allocate, 2-dim array.
```

Allocation

As with all arrays, the new keyword must be used to allocate memory for an array. For example,

Code: Java

```
int[][] a2 =          new int[ 10][ 5];
```

This allocates an int array with 10 rows and 5 columns. As with all objects, the values are initialized to zero (unlike local variables which are uninitialized).

This actually allocates 6 objects: a one-dimensional array of 5 elements for each of the rows, and a one-dimensional array of ten elements, with each element pointing to the appropriate row array.

Processing 2-dimensional arrays

Often 2-dimensional arrays are processed with nested for loops. Notice in the following example how the rows are handled as separate objects

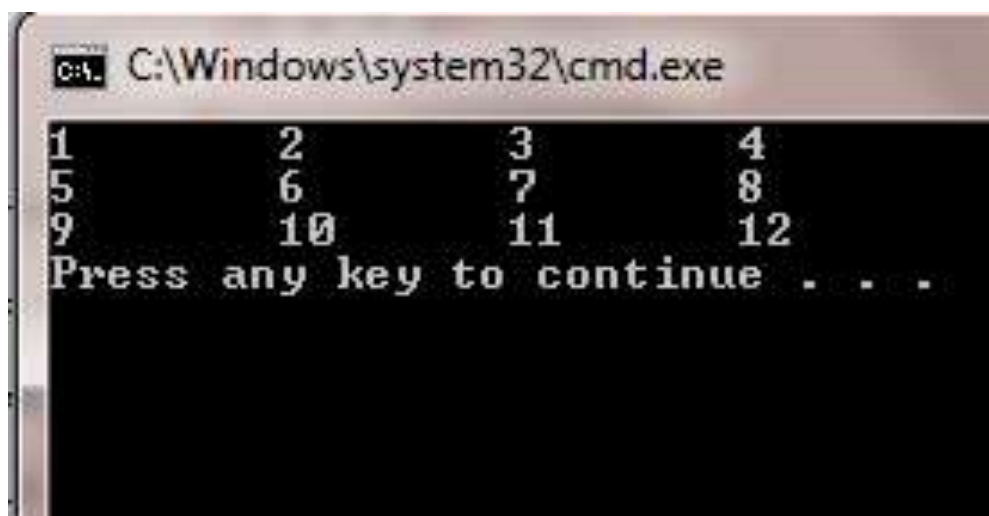
Outer for loop – process the rows

Inner for loop process the columns.

Example 1;

```
class TwoArrayTest{

    public static void main(String arg[])
    {
        int num[ ][]={
{1,2,3,4},
{5,6,7,8},
{9,10,11,12}
        };
int r,c;
        for(r=0;r<3;r++)
        {
            for(c=0;c<4;c++)
            {
                System.out.print(num[r][c]+"\\t");
            }
            System.out.println("");
        }
    }
}
```



Example 2: Two Dimensional Array that allows user keyboard input

```
import java.io.*;

class TwoDimArrayInput{

    public static void main(String args[])throws IOException
    {
        BufferedReader input=new BufferedReader (new InputStreamReader(System.in));
        double num[][]; //array declaration
        int r_size; //variable to specify the row size of the array
        int c_size;//variable to specify the col size of the array
        int maxsize=0;
        System.out.println("Enter the row size of the array:");
        r_size=Integer.parseInt(input.readLine()); //input and parse array row size

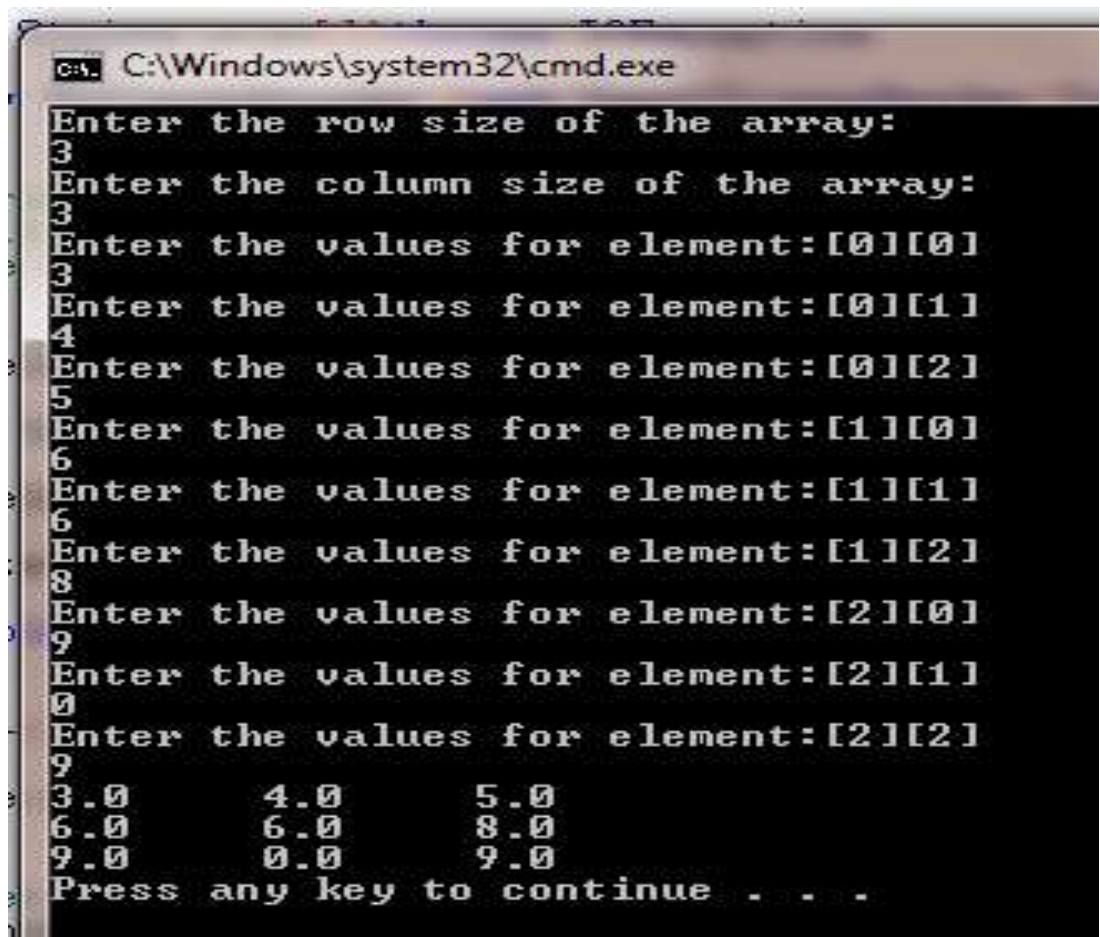
        System.out.println("Enter the column size of the array:");

        c_size=Integer.parseInt(input.readLine()); //input and parse array column size

        num=new double[r_size][c_size]; //allocate array size

        double sum=0.0; double avg=0.0; //declare and initialize sum and average
        for(int r=0;r<r_size;r++) //loop to track the num of value entered
        {
            for(int c=0;c<c_size;c++)
            {
                System.out.println("Enter the values for element:["+r+"]["+c+"]");
                num[r][c]=Double.parseDouble(input.readLine());

            }
        }
        for(int r=0;r<r_size;r++) //loop to track the num of value entered
        {
            for(int c=0;c<c_size;c++)
            {
                System.out.print(num[r][c]+"\\t");
            }
            System.out.println("");
        }
    }
}
```



Example 3 :

```
Class TwoDmatrix{
public static void main(String args[]) {
int twoDm[][]= new int[4][5];
int i,j,k=0;

for(i=0;i<4;i++)
for(j=0;j<5;j++) {
twoDm[i][j]=k;
k++;
}
for(i=0;i<4;i++)
for(j=0;j<5;j++) {
System.out.println(twoDm[i][j]+"" )
System.out.println();
}
}
}
```

Output :

```
C:\Program Files\Java\jdk1.6.0_18\bin>javac TwoDmatrix .java
```

```
C:\Program Files\Java\jdk1.6.0_18\bin>java TwoDmatrix
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

Classes and Objects

Defining Classes

Defining classes is pretty easy; you've seen how to do it a bunch of times in previous lessons. To define a class, use the `class` keyword and the name of the class:

```
class MyClassName {  
...  
}
```

By default, classes inherit from the `Object` class. If this class is a subclass of another specific class (that is, inherits from another class), use `extends` to indicate the superclass of this class:

```
class myClassName extends mySuperClassName {  
...  
}
```

Note

Java 1.1 will give you the ability to nest a class definition inside other classes-a useful construction when you're defining "adapter classes" that implement an interface. The flow of control from the inner class then moves automatically to the outer class

Creating Instance and Class Variables

A class definition with nothing in it is pretty dull; usually, when you create a class, you have something you want to add to make that class different from its superclasses. Inside each class definition are declarations and definitions for variables or methods or both-for the class *and* for each instance. In this section, you'll learn all about instance and class variables; the next section talks about methods.

Defining Instance Variables

Instance variables, fortunately, are declared and defined in almost exactly the same way as local variables; the main difference is their location in the class definition. Variables are considered instance variables if they are declared outside a method definition. Customarily, however, most instance variables are defined

just after the first line of the class definition. For example, Listing 6.1 shows a simple class definition for the class `Bicycle`, which inherits from the class `PersonPoweredVehicle`. This class definition contains five instance variables:

- `bikeType` -The kind of bicycle this bicycle is-for example, `Mountain` or `Street`
- `chainGear` -The number of gears in the front
- `rearCogs` -The number of minor gears on the rear axle
- `currentGearFront` and `currentGearRear` -The gear the bike is currently in, both front and rear

Listing 6.1. The `Bicycle` class.

```
1: class Bicycle extends PersonPoweredVehicle {  
2:     String bikeType;  
3:     int chainGear;  
4:     int rearCogs;  
5:     int currentGearFront;  
6:     int currentGearRear;  
7: }
```

Class Variables

As you have learned in previous lessons, class variables are global to a class and to all that class's instances. You can think of class variables as being even more global than instance variables. Class variables are good for communicating between different objects with the same class, or for keeping track of global states among a set of objects.

To declare a class variable, use the `static` keyword in the class declaration:

```
static int sum;  
static final int maxObjects = 10;
```

Creating Methods

Methods in Object-Oriented Programming and Java," define an object's behavior-what happens when that object is created and the various operations that object can perform during its lifetime. In this section, you'll

get a basic introduction to method definition and how methods work; tomorrow, you'll go into more detail about advanced things you can do with methods.

Defining Methods

Method definitions have four basic parts:

- The name of the method
- The type of object or primitive type the method returns
- A list of parameters
- The body of the method

Note

To keep things simple today, I've left off two optional parts of the method definition: a modifier such as `public` or `private`, and the `throws` keyword, which indicates the exceptions a method can throw

The first three parts of the method definition form what's called the method's *function header* or *signature* and indicate the most important information about the method itself.

In other languages, the name of the method (or function, subroutine, or procedure) is enough to distinguish it from other methods in the program. In Java, you can have different methods that have the same name but a different return type or argument list, so all these parts of the method definition are important. This is called *method overloading*, and you'll learn more about it tomorrow.

New Term

A method's *signature* is a combination of the name of the method, the type of object or primitive data type this method returns, and a list of parameters.

Here's what a basic method definition looks like:

```
returntype    methodname (type1 arg1, type2 arg2, type3 arg3..) {  
...  
}
```

The *return type* is the type of value this method returns. It can be one of the primitive types, a class name, or `void` if the method does not return a value at all.

Example:

```
int computeSum(int x int y)
{
int sum=0;
sum=x+y;
return sum;
}
```

- The method's parameter list is a set of variable declarations, separated by commas, inside parentheses. These parameters become local variables in the body of the method, whose values are the objects or values of primitives passed in when the method is called.
- Inside the body of the method you can have statements, expressions, method calls to other objects, conditionals, loops, and so on-everything you've learned about in the previous lessons.
- If your method has a real return type (that is, it has not been declared to return `void`), somewhere inside the body of the method you need to explicitly return a value. Use the `return` keyword to do this.
- An example below of a class that defines a `makeRange()` method. `makeRange()` takes two integers- a lower bound and an upper bound-and creates an array that contains all the integers between those two boundaries (inclusive).

Example The RangeClass class.

```
1: class RangeClass {
2:     int[] makeRange(int lower, int upper) {
3:         int arr[] = new int[ (upper - lower) + 1 ];
4:
5:         for (int i = 0; i < arr.length; i++) {
6:             arr[i] = lower++;
7:         }
8:         return arr;
9:     }
10:
11:     public static void main(String arg[]) {
12:         int theArray[];
```

```

13:      RangeClass theRange = new RangeClass();
14:
15:      theArray = theRange.makeRange(1, 10);
16:      System.out.print("The array: [ ");
17:      for (int i = 0; i < theArray.length; i++) {
18:          System.out.print(theArray[i] + " ");
19:      }
20:      System.out.println("]");
21:  }
22:
23: }

```

OUTPUT

The array: [1 2 3 4 5 6 7 8 9 10]

Analysis

The `main()` method in this class tests the `makeRange()` method by creating a range where the lower and upper boundaries of the range are 1 and 10, respectively (see line 6), and then uses a `for` loop to print the values of the new array.

The `this` Keyword

In the body of a method definition, you may want to refer to the current object-the object in which the method is contained in the first place-to refer to that object's instance variables or to pass the current object as an argument to another method. To refer to the current object in these cases, you can use the `this` keyword. `this` can be used anywhere the current object might appear-in dot notation to refer to the object's instance variables, as an argument to a method, as the return value for the current method, and so on. Here's an example:

```

t = this.x;           // the x instance variable for this object
this.myMethod(this); // call the myMethod method, defined in
// this class, and pass it the current
// object
return this;          // return the current object

```

In many cases you may be able to omit the `this` keyword entirely. You can refer to both instance variables and method calls defined in the current class simply by name; the `this` is implicit in those references. So the first two examples could be written like this:

```
t = x           // the x instance variable for this object
myMethod()     // call the myMethod method, defined in this
//                                     class
```

Note

Omitting the `this` keyword for instance variables depends on whether there are no variables of the same name declared in the local scope. See the next section for more details on variable scope.

Keep in mind that because `this` is a reference to the current *instance* of a class, you should only use it inside the body of an instance method definition. Class methods—that is, methods declared with the `static` keyword—cannot use `this`.

Variable Scope and Method Definitions

When you declare a variable, that variable always has a limited scope. Variable scope determines where that variable can be used. Variables with a local scope, for example, can only be used inside the block in which they were defined. Instance variables have a scope that extends to the entire class so they can be used by any of the methods within that class.

New Term

Variable scope determines where a variable can be used.

When you refer to a variable within your method definitions, Java checks for a definition of that variable first in the current scope (which may be a block, for example, inside a loop), then in the outer scopes up to the current method definition. If that variable is not a local variable, Java then checks for a definition of that variable as an instance or class variable in the current class, and then, finally, in each superclass in turn.

- Because of the way Java checks for the scope of a given variable, it is possible for you to create a variable in a lower scope such that a definition of that same variable "hides" the original value of that variable. This can introduce subtle and confusing bugs into your code.

For example, note the small Java program in Listing 6.3.

Listing 6.3. A variable scope example.

```
1: class ScopeTest {
2:     int test = 10;
3:
4:     void printTest () {
5:         int test = 20;
6:         System.out.println("test = " + test);
7:     }
8:
9:     public static void main (String args[]) {
10:         ScopeTest st = new ScopeTest();
11:         st.printTest();
12:     }
13: }
```

Analysis

In this class, you have two variables with the same name and definition: The first, an instance variable, has the name `test` and is initialized to the value `10` . The second is a local variable with the same name, but with the value `20` . Because the local variable hides the instance variable, the `println()` method will print that `test` is `20` .

Passing Arguments to Methods

When you call a method with object parameters, the variables you pass into the body of the method are passed by reference, which means that whatever you do to those objects inside the method affects the original objects as well. This includes arrays and all the objects that arrays contain; when you pass an array

into a method and modify its contents, the original array is affected. (Note that primitive types are passed by value.)

Listing 6.4 is an example to demonstrate how this works.

Listing 6.4. The PassByReference class.

```
1: class PassByReference {
2:     int onetoZero(int arg[]) {
3:         int count = 0;
4:
5:         for (int i = 0; i < arg.length; i++) {
6:             if (arg[i] == 1) {
7:                 count++;
8:                 arg[i] = 0;
9:             }
10:        }
11:        return count;
12:    }
13:    public static void main (String arg[]) {
14:        int arr[] = { 1, 3, 4, 5, 1, 1, 7 };
15:        PassByReference test = new PassByReference();
16:        int numOnes;
17:
18:        System.out.print("Values of the array: [ ");
19:        for (int i = 0; i < arr.length; i++) {
20:            System.out.print(arr[i] + " ");
21:        }
22:        System.out.println("]");
23:
24:        numOnes = test.onetoZero(arr);
25:        System.out.println("Number of Ones = " + numOnes);
26:        System.out.print("New values of the array: [ ");
27:        for (int i = 0; i < arr.length; i++) {
28:            System.out.print(arr[i] + " ");
29:        }
30:        System.out.println("]");
31:    }
32:}
```

OUTPUT

Values of the array: [1 3 4 5 1 1 7]

Number of Ones = 3

New values of the array: [0 3 4 5 0 0 7]

Analysis

Note the method definition for the `onetoZero()` method in lines 2 to 12, which takes a single array as an argument. The `onetoZero()` method does two things:

- It counts the number of 1s in the array and returns that value.
- If it finds a 1, it substitutes a 0 in its place in the array.

The `main()` method in the `PassByReference` class tests the use of the `onetoZero()` method. Let's go over the `main()` method line by line so that you can see what is going on and why the output shows what it does.

Lines 14 through 16 set up the initial variables for this example. The first one is an array of integers; the second one is an instance of the class `PassByReference`, which is stored in the variable `test`. The third is a simple integer to hold the number of ones in the array.

Lines 18 through 22 print out the initial values of the array; you can see the output of these lines in the first line of the output.

Line 24 is where the real work takes place; this is where you call the `onetoZero()` method, defined in the object `test`, and pass it the array stored in `arr`. This method returns the number of ones in the array, which you'll then assign to the variable `numOnes`.

Got it so far? Line 25 prints out the number of 1s (that is, the value you got back from the `onetoZero()` method). It returns 3, as you would expect.

The last bunch of lines print out the array values. Because a reference to the array object is passed to the method, changing the array inside that method changes that original copy of the array. Printing out the

values in lines 27 through 30 proves this-that last line of output shows that all the 1 s in the array have been changed to 0 s.

Class Methods

Just as you have class and instance variables, you also have class and instance methods, and the differences between the two types of methods are analogous. Class methods are available to any instance of the class itself and can be made available to other classes. Therefore, some class methods can be used anywhere, regardless of whether an instance of the class exists.

For example, the Java class libraries include a class called `Math` . The `Math` class defines a whole set of math operations that can be used in any program or the various number types:

```
float root = Math.sqrt(453.0);
System.out.print("The larger of x and y is " + Math.max(x, y));
```

To define class methods, use the `static` keyword in front of the method definition, just as you would create a class variable. For example, that `max` class method might have a signature like this:

```
static int max(int arg1, int arg2) { ... }
```

Java supplies "wrapper" classes for each of the primitive data types-for example, classes for `Integer` , `Float` , and `boolean` . Using class methods defined in those classes, you can convert to and from objects and primitive types. For example, the `parseInt()` class method in the `Integer` class takes a string and a radix (base) and returns the value of that string as an integer:

```
int count = Integer.parseInt("42", 10) // returns 42
```

Most methods that operate on a particular object, or that affect that object, should be defined as instance methods. Methods that provide some general utility but do not directly affect an instance of that class are better declared as class methods.

Creating Object

- Object is an instance of a class.
- It is any entity that has state and behavior.
- Object is created from class. Then it is used to access class members – variables & methods.

- Object is created in main method.

Consider

```
class Customer{
// declare member variables
String custID;
String Names;
String Address;
String Prod;
void getdetails()
{
// code here
}
Void computeOrderValue ()
{
//code here
}
```

To create object of type customer then,

```
public static void main(String args[])
{
Customer MyCust=new Customer(); // create object myCust
```

The object need to be initialized using - constructor, **new Customer();**

To create many objects use,

```
public static void main(String args[])
{
Customer MyCust=new Customer();
Customer cust2= new Customer();
```

Using object

The object is use to call member methods, a mechanism call message passing.

Syntax : objectName.MemberMethod();

e.g

```
MyCust.getDetails(); //use object to call member methods
MyCust.computeOrderValue(20,7000.00);
```

It is also used to access member variables

Syntax : objectName.MemberVariable;

Mycust.custID=1;

Examples of Class programs

Example 1

```
class Customer{
// declare member variables
String custID;
String Names;
String Address;
String Prod;

// declare and implement class methods
void getDetails()
{
custID="C001";
Names="Xcom LTD";
Address="Nairobi";
Prod="TV";

System.out.println("Customer ID="+custID);
System.out.println("Names =" +Names);
System.out.println("Customer Address="+Address);
System.out.println("Product Name="+Prod);
}
void computeOrderValue(int qty, double cost)
{
double order_val=0.0;
order_val=qty*cost;
System.out.println("Total Oder Value="+order_val);
}
public static void main(String args[])
{
Customer MyCust=new Customer(); // create object myCust
MyCust.getDetails(); //use object to call member methods
MyCust.computeOrderValue(20,7000.00);
}
}
```

Example 2

```
class RectArea{
```

```

int leg;
int wid;

int getRectArea(int l,int w)
{
leg=l;
wid=w;
int Area=0;
Area=leg*wid;
return Area;
}

public static void main(String args[])
{
RectArea myArea=new RectArea();

int return_Area;
return_Area=myArea.getRectArea(30,10);

System.out.println("The Area="+return_Area+"sq.m");
}
}

```

Overloading methods

- Method overloading refers to creating methods with the same name but different arguments, or return data type.
- Method overloading allows instances of your class to have a simpler interface to other objects (no need for entirely different methods with different names that do essentially the same thing) and to behave differently based on the input to that method.
- For example, an overloaded `draw()` method could be used to draw just about anything, whether it were a circle or a point or an image. The same method name, with different arguments, could be used for all cases.

```

void draw()
{
//code here
}
int draw(int x, int y)
{
//code here
}
double draw(double a, double d, double c)
{
//code here
}

```

```
}
```

Draw is an overloaded method.

- When you call a method in an object, Java matches up the method name and the number and type of arguments to choose which method definition to execute.

New Term

Method overloading is creating multiple methods with the same name but with different signatures and definitions. Java uses the number and type of arguments to choose which method definition to execute.

- To create an overloaded method, all you need to do is create several different method definitions in your class, all with the same name, but with different parameter lists (either in number or type of arguments). Java allows method overloading as long as each parameter list is unique for the same method name.
- Note that Java differentiates overloaded methods based on the number and type of parameters to that method, not on the method's return type. That is, if you try to create two methods with the same name and same parameter list, but different return types, you'll get a compiler error. Also, the variable names you choose for each parameter to the method are irrelevant—all that matters is the number and the type.
- Here's an example of creating an overloaded method. Listing 7.1 shows a simple class definition for a class called `MyRect`, which defines a rectangular shape. The `MyRect` class has four instance variables to define the upper-left and lower-right corners of the rectangle: `x1`, `y1`, `x2`, and `y2`.

Note

Why did I call it `MyRect` instead of just `Rectangle`? The `java.awt` package has a class called `Rectangle` that implements much of this same behavior. I called this class `MyRect` to prevent confusion between the two classes.

Listing 7.1. The `MyRect` class.

```
1: class MyRect {  
2:     int x1 = 0;  
3:     int y1 = 0;
```

```
4:    int x2 = 0;
5:    int y2 = 0;
6: }
```

Note

Don't try to compile this example yet. Actually, it'll compile just fine, but it won't run because it doesn't (yet) have a `main()` method. When you're finished building this class definition, the final version can be compiled and run.

When a new instance of the `myRect` class is initially created, all its instance variables are initialized to `0`. Let's define a `buildRecpt()` method that takes four integer arguments and "resizes" the rectangle to have the appropriate values for its corners, returning the resulting rectangle object (note that because the arguments have the same names as the instance variables, you have to make sure to use `this` to refer to them):

```
MyRect buildRect(int x1, int y1, int x2, int y2) {
this.x1 = x1;
this.y1 = y1;
this.x2 = x2;
this.y2 = y2;
return this;
}
```

What if you want to define a rectangle's dimensions in a different way-for example, by using `Point` objects rather than individual coordinates? You can overload `buildRect()` so that its parameter list takes two `Point` objects (note that you'll also need to import the `java.awt.Point` class at the top of your source file so Java can find it):

```
MyRect buildRect(Point topLeft, Point bottomRight) {
x1 = topLeft.x;
y1 = topLeft.y;
x2 = bottomRight.x;
y2 = bottomRight.y;
return this;
}
```

Perhaps you want to define the rectangle using a top corner and a width and height. You can do that, too.

Just create a different definition for `buildRect()` :

```
MyRect buildRect(Point topLeft, int w, int h) {
    x1 = topLeft.x;
    y1 = topLeft.y;
    x2 = (x1 + w);
    y2 = (y1 + h);
    return this;
}
```

To finish up this example, let's create a method-called `printRect()` -to print out the rectangle's coordinates, and a `main()` method to test it all (just to prove that this does indeed work). Listing 7.2 shows the completed class definition with all its methods: three `buildRect()` methods, one `printRect()` , and one `main()` .

The complete `MyRect` class.

```
1:import java.awt.Point;
2:
3:class MyRect {
4:    int x1 = 0;
5:    int y1 = 0;
6:    int x2 = 0;
7:    int y2 = 0;
8:
9:    MyRect buildRect(int x1, int y1, int x2, int y2) {
10:        this.x1 = x1;
11:        this.y1 = y1;
12:        this.x2 = x2;
13:        this.y2 = y2;
14:        return this;
15:    }
16:
17:    MyRect buildRect(Point topLeft, Point bottomRight) {
18:        x1 = topLeft.x;
19:        y1 = topLeft.y;
20:        x2 = bottomRight.x;
21:        y2 = bottomRight.y;
22:        return this;
23:    }
24:
25:    MyRect buildRect(Point topLeft, int w, int h) {
26:        x1 = topLeft.x;
27:        y1 = topLeft.y;
28:        x2 = (x1 + w);
29:        y2 = (y1 + h);
30:        return this;

```

```

31:     }
32:
33:     void printRect(){
34:         System.out.print("MyRect: <" + x1 + ", " + y1);
35:         System.out.println(", " + x2 + ", " + y2 + ">");
36:     }
37:
38:     public static void main(String args[]) {
39:         MyRect rect = new MyRect();
40:
41:         System.out.println("Calling buildRect with coordinates 25,25
42:         50,50:");
43:         rect.buildRect(25, 25, 50, 50);
44:         rect.printRect();
45:         System.out.println("-----");
46:
47:         System.out.println("Calling buildRect w/points (10,10), (20,20):");
48:         rect.buildRect(new Point(10,10), new Point(20,20));
49:         rect.printRect();
50:         System.out.println("-----");
51:
52:         System.out.print("Calling buildRect w/1 point (10,10),");
53:         System.out.println(" width (50) and height (50):");
54:
55:         rect.buildRect(new Point(10,10), 50, 50);
56:         rect.printRect();
57:         System.out.println("-----");
58:     }

```

OUTPUT

Calling buildRect with coordinates 25,25 50,50:

MyRect: <25, 25, 50, 50>

Calling buildRect w/points (10,10), (20,20):

MyRect: <10, 10, 20, 20>

Calling buildRect w/1 point (10,10), width (50) and height (50):

MyRect: <10, 10, 60, 60>

As you can see from this example, all the `buildRect()` methods work based on the arguments with which they are called. You can define as many versions of a method as you need to in your own classes to implement the behavior you need for that class.

Constructor Methods

- In addition to regular methods, you can also define constructor methods in your class definition.
- Constructor methods are used to initialize new objects when they're created.
- Unlike regular methods, you can't call a constructor method by calling it directly; instead, constructor methods are called by Java automatically when you create a new object.
 1. Constructor performs the following functions:
 2. Allocates memory for the new object
 3. Initializes that object's instance variables, either to their initial values or to a default (`0` for numbers, `null` for objects, `false` for booleans, `'\0'` for characters)
 4. Calls the class's constructor method (which may be one of several methods)

New Term

Constructor methods are special methods that are called automatically by Java to initialize a new object.

- If a class doesn't have any special constructor methods defined, you'll still end up with a new object, but you might have to set its instance variables or call other methods that the object needs to initialize itself. All the examples you've created up to this point have behaved like this.
- By defining constructor methods in your own classes, you can set initial values of instance variables, call methods based on those variables or on other objects, or calculate initial properties of your object.
- You can also overload constructors, as you would regular methods, to create an object that has specific properties based on the arguments you give in the `new` expression.

Basic Constructors

Constructors look a lot like regular methods, with two basic differences:

- Constructors always have the same name as the class.
- Constructors don't have a return type.
-

Example below shows a simple class called `Person`. The constructor method for `Person` takes two arguments: a string object representing a person's name and an integer for the person's age.

The Person class.

```
1: class Person {
2:     String name;
3:     int age;
4:
5:     Person(String n, int a) // constructor method
6:     {
7:         name = n;
8:         age = a;
9:     }
10:    void printPerson() {
11:        System.out.print("Hi, my name is " + name);
12:        System.out.println(". I am " + age + " years old.");
13:    }
14:
15:    public static void main (String args[]) {
16:        Person p;
17:        p = new Person("Laura", 20);
18:        p.printPerson();
19:        System.out.println("-----");
20:        p = new Person("Tommy", 3);
21:        p.printPerson();
22:        System.out.println("-----");
23:    }
24:}
```

OUTPUT

Hi, my name is Laura. I am 20 years old.

Hi, my name is Tommy. I am 3 years old.

- The person class has three methods: The first is the constructor method, defined in lines 5 to 8, which initializes the class's two instance variables based on the arguments to `new` . The `Person` class also includes a method called `printPerson()` so that the object can "introduce" itself, and a `main()` method to test each of these things.

Another

```
class RectArea{
```

```
int leg;
```

```
int wid;
```

```
RectArea ()//constructor method
```

```
{
```

```
leg=0;
```

```
wid=0;
```

```
}
```

```
int getRectArea(int l,int w)
```

```

{
leg=l;
wid=w;
int Area=0;
Area=leg*wid;
return Area;
}

public static void main(String args[])
{
RectArea myArea=new RectArea();

int return_Area;
return_Area=myArea.getRectArea(30,10);

System.out.println("The Area="+return_Area+"sq.m");
}
}

```

Overloading Constructors

- Like regular methods, constructors can also take varying numbers and types of parameters, enabling you to create your object with exactly the properties you want it to have, or for it to be able to calculate properties from different kinds of input.
- For example, the `buildRect()` methods you defined in the `MyRect` class earlier today would make excellent constructors because they're initializing an object's instance variables to the appropriate values. So, for example, instead of the original `buildRect()` method you had defined (which took four parameters for the coordinates of the corners), you could create a constructor instead.
- Listing below shows a new class, `MyRect2`, that has all the same functionality of the original `MyRect`, except with overloaded constructor methods instead of the overloaded `buildRect()` method. The output shown at the end is also the same output as for the previous `MyRect` class; only the code to produce it has changed.

The `MyRect2` class (with constructors).

```

1: import java.awt.Point;
2:
3: class MyRect2 {
4:     int x1 = 0;
5:     int y1 = 0;
6:     int x2 = 0;
7:     int y2 = 0;
8:
9:     MyRect2(int x1, int y1, int x2, int y2) {
10:         this.x1 = x1;

```

```

11:         this.y1 = y1;
12:         this.x2 = x2;
13:         this.y2 = y2;
14:     }
15:
16:     MyRect2(Point topLeft, Point bottomRight) {
17:         x1 = topLeft.x;
18:         y1 = topLeft.y;
19:         x2 = bottomRight.x;
20:         y2 = bottomRight.y;
21:     }
22:
23:     MyRect2(Point topLeft, int w, int h) {
24:         x1 = topLeft.x;
25:         y1 = topLeft.y;
26:         x2 = (x1 + w);
27:         y2 = (y1 + h);
28:     }
29:
30:     void printRect() {
31:         System.out.print("MyRect: <" + x1 + ", " + y1);
32:         System.out.println(", " + x2 + ", " + y2 + ">");
33:     }
34:
35:     public static void main(String args[]) {
36:         MyRect2 rect;
37:
38:         System.out.println("Calling MyRect2 with coordinates 25,25 50,50:");
39:         rect = new MyRect2(25, 25, 50,50);
40:         rect.printRect();
41:         System.out.println("-----");
42:
43:         System.out.println("Calling MyRect2 w/points (10,10), (20,20):");
44:         rect= new MyRect2(new Point(10,10), new Point(20,20));
45:         rect.printRect();
46:         System.out.println("-----");
47:
48:         System.out.print("Calling MyRect2 w/1 point (10,10)");
49:         System.out.println(" width (50) and height (50):");
50:         rect = new MyRect2(new Point(10,10), 50, 50);
51:         rect.printRect();
52:         System.out.println("-----");
53:
54:     }
55: }

```

OUTPUT

Calling MyRect2 with coordinates 25,25 50,50:
MyRect: <25, 25, 50, 50>

Calling MyRect2 w/points (10,10), (20,20):
MyRect: <10, 10, 20, 20>

Calling MyRect2 w/1 point (10,10), width (50) and height (50):
MyRect: <10, 10, 60, 60>

Class Inheritance

- Inheritance is involves creating new classes from the exiting classes.
- The new classes acquire the behavior of the parent class in addition to its own new features.
- Terms : Parent, Base class, super class – Refers to original class from which new classes are created.
- Term – Child, derived, sub class - refers to new created classes.

Creating new sub class

- In Java inheritance is implemented by use of keyword **extends**

Syntax

```
class SubClassName extendsBaseName {  
    member variables;  
    Member methods();  
}
```

Example

```
class Person{ // base class  
  
    String IDNo;  
    String Names;  
    int Age;  
  
    void getPersonDetails()  
    {  
        IDNo="2313333";  
        Names="john were";  
        Age=30;  
    }  
  
    void DisplayPersonDetails()  
    {  
        System.out.println("The Id Number is: "+IDNo);  
        System.out.println("The Name is: "+Names);  
        System.out.println("The age  is: "+Age);  
    }  
  
} // End of base class  
  
class Student extends Person{ // Create subclass  
    String Course;  
    int Duration;  
    double Fee;
```

```

void getStuduntDetails()
{
    getPersonDetails(); // call the base class method
    Course="BIT";
    Duration=3;
    Fee=50000.00;
}

DisplayStudentDetails()
{
    DisplayPersonDetails(); // call the base class method
    System.out.println("The Id course is: "+Course);
    System.out.println("The course Duration is : "+Duration);
    System.out.println("The Sem Fee is: "+Fee);
}

}

public static void main(String args[])
{
    Student john=new Student();
    john.getStuduntDetails();
    john.DisplayStudentDetails();
}

}

```

The base class person consists of

Member variables : IDNo, Names, Age

Member functions : getPersonDetails(),DisplayPersonDetails()

The sub class student consists of

Member variables : Course,Duration, Fee

Member functions : getStuduntDetails(),DisplayStudentDetails()

When the class student is implemented, it only implements the unique details which are not available in the person class. The details about the person which the student class needs then are inherited from the parent person class.

NB:

1. Save the file using sub class name – student
2. Include only one main method in the student class.

Food for thought.

- Explain the advantages of inheritance.
- Explain the disadvantages of inheritance.
- Explain the Java inheritance restrictions
- Differentiate between Single and multiple inheritance. Does Java support the two types of inheritance?.Explain.

Overriding Constructors

- Because constructors have the same name as the current class, you cannot technically override a superclass's constructors. If you want a constructor in a subclass with the same number and type of arguments as in the superclass, you'll have to define that constructor in your own class.
- However, when you create your constructors you will almost always want to call your superclass's constructors to make sure that the inherited parts of your object get initialized the way your superclass intends them to be. By explicitly calling your superclasses constructors in this way you can create constructors that effectively override or overload your superclass's constructors.
- To call a regular method in a superclass, you use the form `super.methodname(arguments)` .
Because with constructors you don't have a method name to call, you have to use a different form:

`super(arg1, arg2, ...);`

- Note that Java has a specific rule for the use of `super()` : It must be the very first thing in your constructor definition. If you don't call `super()` explicitly in your constructor, Java will do it for you-using `super()` with no arguments.
- Similar to using `this(...)` in a constructor, `super(...)` calls a constructor method for the immediate superclass with the appropriate arguments (which may, in turn, call the constructor of its superclass, and so on).
- Note that a constructor with that signature has to exist in the superclass in order for the call to `super()` to work. The Java compiler will check this when you try to compile the source file.
- Note that you don't have to call the constructor in your superclass that has exactly the same signature as the constructor in your class; you only have to call the constructor for the values you need initialized. In fact, you can create a class that has constructors with entirely different signatures from any of the superclass's constructors.

Listing below shows a class called `NamedPoint`, which extends the class `Point` from Java's `awt` package. The `Point` class has only one constructor, which takes an `x` and a `y` argument and returns a `Point` object. `NamedPoint` has an additional instance variable (a string for the name) and defines a constructor to initialize `x`, `y`, and the name.

Listing 7.9. The `NamedPoint` class.

```
1: import java.awt.Point;
2: class NamedPoint extends Point {
3:     String name;
4:
5:     NamedPoint(int x, int y, String name) {
6:         super(x,y);
7:         this.name = name;
8:     }
9:     public static void main (String arg[]) {
10:         NamedPoint np = new NamedPoint(5, 5, "SmallPoint");
11:         System.out.println("x is " + np.x);
12:         System.out.println("y is " + np.y);
13:         System.out.println("Name is " + np.name);
14:     }
15: }
```

OUTPUT

```
x is 5
y is 5
name is SmallPoint
```

The constructor defined here for `NamedPoint` (lines 5 through 8) calls `Point`'s constructor method to initialize `Point`'s instance variables (`x` and `y`). Although you can just as easily initialize `x` and `y` yourself, you may not know what other things `Point` is doing to initialize itself, so it's always a good idea to pass constructors up the hierarchy to make sure everything is set up correctly.

Finalizer Methods

- Finalizer methods are almost the opposite of constructor methods; whereas a constructor method is used to initialize an object, finalizer methods are called just before the object is garbage-collected and its memory reclaimed.
- The finalizer method is named simply `finalize()`. The `Object` class defines a default finalizer method, which does nothing. To create a finalizer method for your own classes, override the `finalize()` method using this signature:


```
protected void finalize() throws Throwable {  
    super.finalize();  
}
```

Note

The `throws Throwable` part of this method definition refers to the errors that might occur when this method is called. Errors in Java are called exceptions

- Inside the body of that `finalize()` method, include any cleaning up you want to do for that object. You can also call `super.finalize()` to allow your class's superclasses to finalize your object, if necessary (it's a good idea to do so just to make sure that everyone gets a chance to deal with the object if they need to).
- You can always call the `finalize()` method yourself at any time; it's just a plain method like any other. However, calling `finalize()` does not trigger an object to be garbage-collected. Only removing all references to an object will cause it to be marked for deleting.
- Finalizer methods are best used for optimizing the removal of an object—for example, by removing references to other objects, by releasing external resources that have been acquired (for example, external files), or for other behaviors that may make it easier for that object to be removed. In most cases, you will not need to use `finalize()` at all.

Java Applets

- Java applications are standalone Java programs that can be run by using just the Java interpreter, for example, from a command line.
- Java applets run from inside a World Wide Web browser. They are embedded on Web page as image is done.
- A reference to an applet embedded in a Web page is done using a special HTML tag.
- When a reader, using a Java-enabled browser, loads a Web page with an applet in it, the browser downloads that applet from a Web server and executes it on the local system (the one the browser is running on).
- The Java interpreter is built into the browser and runs the compiled Java class file from there.

Java Applet Security Restrictions

There is the set of restrictions placed on how applets can operate in the name of security. Given the fact that Java applets can be downloaded from any site on the World Wide Web and run on a client's system, Java-enabled browsers and tools limit what can be done to prevent a rogue applet from causing system damage or security breaches. Without these restrictions in place, Java applets could be written to contain viruses or Trojan horses (programs that seem friendly but do some sort of damage to the system), or be used to compromise the security of the system that runs them. The restrictions on applets include the following:

Applets **can't read or write to the reader's file system** , which means they cannot delete files or test to see what programs you have installed on the hard drive.

Applets **can't communicate with any network server** other than the one that had originally stored the applet, to prevent the applet from attacking another system from the reader's system.

Applets **can't run any programs on the reader's system** . For UNIX systems, this includes forking a process.

Applets **can't load programs native to the local platform** , including shared libraries such as DLLs.

Creating Applets

- To create an applet, you create a subclass of the class **Applet** .
- The `Applet` class, part of the `java.applet` package, provides much of the behavior your applet needs to work inside a Java-enabled browser.

Therefore need to import the package as import **java.applet. Applet;**

Then extend the class as follows

public class MyApplet extends Applet{

Or

public class MyApplet extends java.applet.Applet {

Applets also take strong advantage of Java's **Abstract Windowing Toolkit (awt)**, which provides behavior for creating graphical user interface (GUI)-based applets and applications:

- o drawing to the screen; creating windows,
- o menu bars,
- o buttons,
- o check boxes, and
- o other UI elements;

AWT is also used for managing user input such as mouse clicks and keypresses. The awt classes are part of the java.awt package.

New Term

Java's Abstract Windowing Toolkit (awt) provides classes and behavior for creating GUI-based applications in Java. Applets make use of many of the capabilities in the awt.

Need to import it.

import java.awt.Graphics;

import java.awt.Font;

import java.awt.Color;

or

import java.awt.*;

Example : first applet code

import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("This is my first applet", 60,40);
}
}

NB: Applets do not use main () method. This is because the applets are loaded when a web page is started.

Creating Web page to attach applet code.

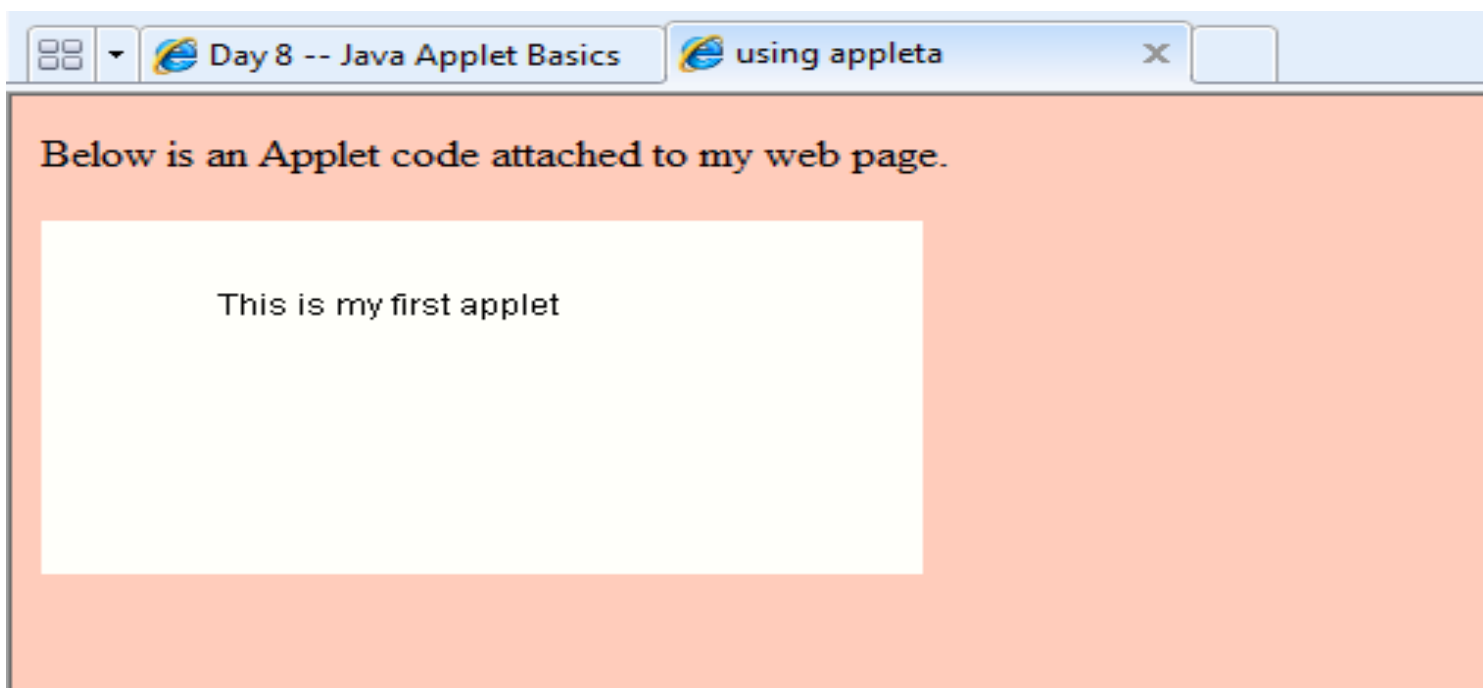
```
<html>
<head>
<title> using applets</title>
</head>
<body>
<p> Below is an Applet code attached to my web page.</p>
<applet code="MyApplet.class" width=300 eight=150></applet>
</body>
</html>
```

Compiling the applet source code

Javac MyApplet.java

To run the applet on web page:

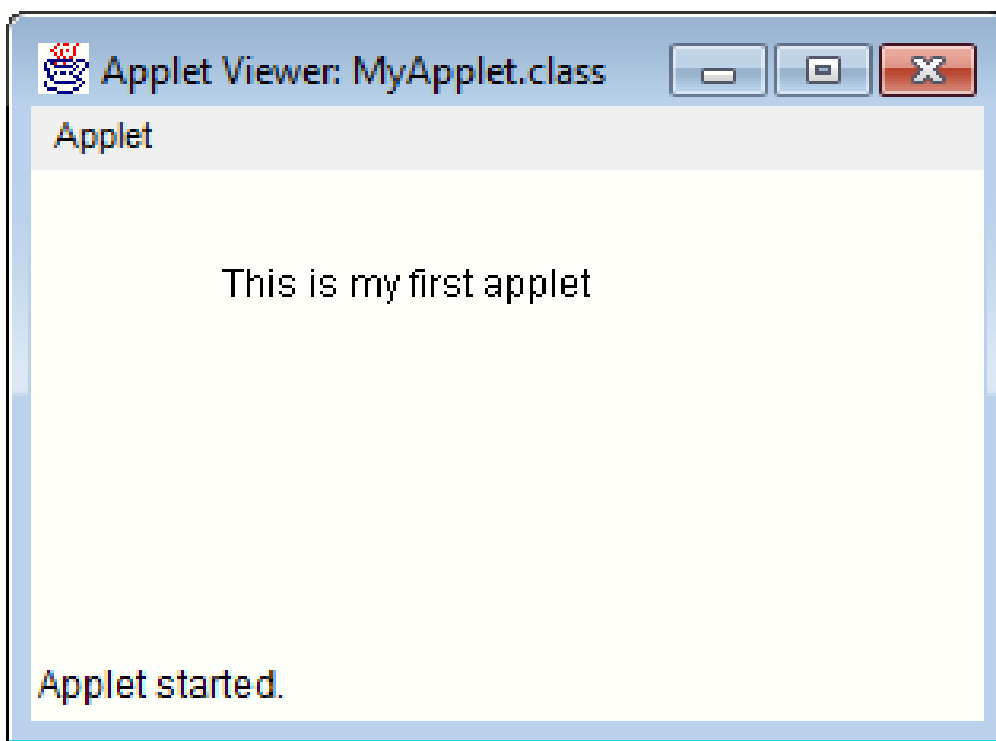
- Start the web page on which the applet is attached



To run applet alone – use the appletviewer tool on command prompt.

If using graphical interface e.g Textpad or Jcreator, then use Tools > Run Java applet.

Appletviewer myApplet.html



Explaining Applet code;

Public void paint(Graphics g)

```
{  
}
```

- Painting- this is how applet draws something on the screen – text, line or image.
- Paint Takes argument – of type graphics instance.
- Object graphics holds graphics state – color, font.

g.drawString("This is my first applet", 60,40);

provides – (x,y) coordinates. It specifies the position to start printing the applet text on the screen.

60 – Pixels from left

40 – Pixels from top.

Understanding <APPLET> tag

<applet code="MyApplet.class" width=300 eight=150></applet>

code="MyApplet.class" – specify the class file generated after compilation. It is the class file which is attached on the web page for execution.

width=300 height=150 – specifies the dimension of the size of the applet to occupy on the web page

Using code base

It is used when the class files and the web pages to attach the applet are not in the same folder.

Codebase – specifies the sub folder containing the class files.

```
<applet code="MyApplet.class" codebase="myclassfiles" width=300
height=150></applet>
```

Major Applet Activities

- To create a basic Java application, your class has to have one method, `main()`, with a specific signature. Then, when your application runs, `main()` is found and executed, and from `main()` you can set up the behavior that your program needs to run. Applets are similar but more complicated- and, in fact, applets don't need a `main()` method at all. Applets have many different activities that correspond to various major events in the life cycle of the applet- for example, initialization, painting, and mouse events. Each activity has a corresponding method, so when an event occurs, the browser or other Java-enabled tool calls those specific methods.
- The default implementations of these activity methods do nothing; to provide behavior for an event you must override the appropriate method in your applet's subclass. You don't have to override all of them, of course; different applet behavior requires different methods to be overridden.
- The five most important methods in an applet's execution: initialization, starting, stopping, destroying, and painting.

Initialization

- Initialization occurs when the applet is first loaded (or reloaded), similarly to the `main()` method in applications. The initialization of an applet might include reading and parsing any parameters to the applet, creating any helper objects it needs, setting up an initial state, or loading images or fonts. To provide behavior for the initialization of your applet, override the `init()` method in your applet class:

```
public void init() {
    ...
}
```

Starting

- After an applet is initialized, it is started. Starting is different from initialization because it can happen many different times during an applet's lifetime, whereas initialization happens only once. Starting can also occur if the applet was previously stopped. For example, an applet is stopped if the reader follows a link to a different page, and it is started again when the reader returns to this page.

To provide startup behavior for your applet, override the `start()` method:

```
public void start() {  
    ...  
}
```

- Functionality that you put in the `start()` method might include creating and starting up a thread to control the applet, sending the appropriate messages to helper objects, or in some way telling the applet to begin running

Stopping

- Stopping and starting go hand in hand. Stopping occurs when the reader leaves the page that contains a currently running applet, or you can stop the applet yourself by calling `stop()`. By default, when the reader leaves a page, any threads the applet had started will continue running. By overriding `stop()`, you can suspend execution of these threads and then restart them if the applet is viewed again:

```
public void stop() {  
    ...  
}
```

Destroying

- Destroying sounds more violent than it is. Destroying enables the applet to clean up after itself just before it is freed or the browser exits—for example, to stop and remove any running threads, close any open network connections, or release any other running objects. Generally, you won't want to override `destroy()` unless you have specific resources that need to be released—for example, threads that the applet has created. To provide clean-up behavior for your applet, override the

`destroy()` method:

```
public void destroy() {  
    ...  
}
```

Technical Note

How is `destroy()` different from `finalize()`. First, `destroy()` applies only to applets. `finalize()` is a more general-purpose way for a single object of any type to clean up after itself.

Painting

- Painting is how an applet actually draws something on the screen, be it text, a line, a colored background, or an image. Painting can occur many thousands of times during an applet's life cycle (for example, after the applet is initialized, if the browser is placed behind another window on the screen and then brought forward again, if the browser window is moved to a different position on the screen, or perhaps repeatedly, in the case of animation).
- You override the `paint()` method if your applet needs to have an actual appearance on the screen (that is, most of the time). The `paint()` method looks like this:

```
public void paint(Graphics g) {  
    ...  
}
```

- Note that unlike the other major methods in this section, `paint()` takes an argument, an instance of the class `Graphics`. This object is created and passed to `paint` by the browser, so you don't have to worry about it. However, you will have to make sure that the `Graphics` class (part of the `java.awt` package) gets imported into your applet code, usually through an `import` statement at the top of your Java file:

```
import java.awt.Graphics;
```

A Simple Applet : applet to create a string text and apply font face, bold and size.

Listing : The Hello Again applet.

```
1: import java.awt.Graphics;  
2: import java.awt.Font;  
3: import java.awt.Color;  
4:  
5: public class HelloAgainApplet extends java.applet.Applet {  
6:  
7:     Font f = new Font("TimesRoman", Font.BOLD, 36);  
8:  
9:     public void paint(Graphics g) {  
10:         g.setFont(f);  
11:         g.setColor(Color.red);  
12:         g.drawString("Hello again!", 5, 40);  
13:     }  
14: }
```


Passing Parameters to Applets

With Java applications, you pass parameters to your `main()` routine by using arguments on the command line, or, for Macintoshes, in the Java Runner's dialog box. You can then parse those arguments inside the body of your class, and the application acts accordingly, based on the arguments it is given.

Applets, however, don't have a command line. How do you pass in different arguments to an applet?

Applets can get different input from the HTML file that contains the `<APPLET>` tag through the use of applet parameters. To set up and handle parameters in an applet, you need two things:

- A special parameter tag in the HTML file
- Code in your applet to parse those parameters

Applet parameters come in two parts: a parameter name, which is simply a name you pick, and a value, which is the actual value of that particular parameter. So, for example, you can indicate the color of text in an applet by using a parameter with the name `color` and the value `red`. You can determine an animation's speed using a parameter with the name `speed` and the value `5`.

In the HTML file that contains the embedded applet, you indicate each parameter using the `<PARAM>` tag, which has two attributes for the name and the value, called (surprisingly enough) `NAME` and `VALUE`. The `<PARAM>` tag goes inside the opening and closing `<APPLET>` tags:

```
<APPLET CODE="MyApplet.class" WIDTH=100 HEIGHT=100>
<PARAM NAME=font VALUE="TimesRoman">
<PARAM NAME=size VALUE="36">
A Java applet appears here.</APPLET>
```

This particular example defines two parameters to the `MyApplet` applet: one whose name is `font` and whose value is `TimesRoman`, and one whose name is `size` and whose value is `36`.

Parameters are passed to your applet when it is loaded. In the `init()` method for your applet, you can then get hold of those parameters by using the `getParameter()` method. `getParameter()` takes one argument—a string representing the name of the parameter you're looking for—and returns a string containing the corresponding value of that parameter. (Like arguments in Java applications, all the parameter values are strings.) To get the value of the `font` parameter from the HTML file, you might have a line such as this in your `init()` method:

```
String theFontName = getParameter("font");
```

Note

The names of the parameters as specified in `<PARAM>` and the names of the parameters in `getParameter()` must match identically, including having the same case. In other words, `<PARAM NAME="name">` is different from `<PARAM NAME="Name">`. If your parameters are not being properly passed to your applet, make sure the parameter cases match.

Note that if a parameter you expect has not been specified in the HTML file, `getParameter()` returns `null`. Most often, you will want to test for a `null` parameter in your Java code and supply a reasonable default:

```
if (theFontName == null)
theFontName = "Courier"
```

Keep in mind that `getParameter()` returns strings-if you want a parameter to be some other object or type, you have to convert it yourself. To parse the `size` parameter from that same HTML file and assign it to an integer variable called `theSize`, you might use the following lines:

```
int theSize;
String s = getParameter("size");
if (s == null)
theSize = 12;
else theSize = Integer.parseInt(s);
```

Get it? Not yet? Let's create an example of an applet that uses this technique. You'll modify the Hello Again applet so that it says hello to a specific name, for example, `"Hello Bill"` or `"Hello Alice"`. The name is passed into the applet through an HTML parameter.

Let's start by copying the original `HelloAgainApplet` class and calling it `MoreHelloAgain` (see Listing below).

Listing :The More Hello Again applet.

```
1:import java.awt.Graphics;
2:import java.awt.Font;
3:import java.awt.Color;
4:
5:public class MoreHelloApplet extends java.applet.Applet {
6:
7:    Font f = new Font("TimesRoman", Font.BOLD, 36);
8:
```

```
9: public void paint(Graphics g) {
10:     g.setFont(f);
11:     g.setColor(Color.red);
12:     g.drawString("Hello Again!", 5, 40);
13: }
14: }
```

The first thing you need to add to this class is a place to hold the name of the person you're saying hello to. Because you'll need that name throughout the applet, let's add an instance variable for the name, just after the variable for the font in line 7:

```
String name;
```

To set a value for the name, you have to get that parameter from the HTML file. The best place to handle parameters to an applet is inside an `init()` method. The `init()` method is defined similarly to `paint()` (public, with no arguments, and a return type of `void`). Make sure when you test for a parameter that you test for a value of `null`. The default, in this case, if a name isn't indicated, is to say hello to "Laura". Add the `init()` method in between your instance variable definitions and the definition for `paint()`, just before line 9:

```
public void init() {
name = getParameter("name");
if (name == null)
name = "Laura";
}
```

Now that you have the name from the HTML parameters, you'll need to modify it so that it's a complete string—that is, to tack the word `Hello` with a space onto the beginning, and an exclamation point onto the end. You could do this in the `paint()` method just before printing the string to the screen, but that would mean creating a new string every time the applet is painted. It would be much more efficient to do it just once, right after getting the name itself, in the `init()` method. Add this line to the `init()` method just before the last brace:

```
name = "Hello " + name + "!";
```

And now, all that's left is to modify the `paint()` method to use the new name parameter. The original `drawString()` method looked like this:

```
g.drawString("Hello Again!", 5, 40);
```

To draw the new string you have stored in the `name` instance variable, all you need to do is substitute that variable for the literal string:

```
g.drawString(name, 5, 40);
```

Listing 8.4 shows the final result of the `MoreHelloApplet` class. Compile it so that you have a class file ready.

Listing 8.4. The `MoreHelloApplet` class.

```
1: import java.awt.Graphics;
2: import java.awt.Font;
3: import java.awt.Color;
4:
5: public class MoreHelloApplet extends java.applet.Applet {
6:
7:     Font f = new Font("TimesRoman", Font.BOLD, 36);
8:     String name;
9:
10:    public void init() {
11:        name = getParameter("name");
12:        if (name == null)
13:            name = "Laura";
14:
15:        name = "Hello " + name + "!";
16:    }
17:
18:    public void paint(Graphics g) {
19:        g.setFont(f);
20:        g.setColor(Color.red);
21:        g.drawString(name, 5, 40);
22:    }
23: }
```

Now let's create the HTML file that contains this applet. Listing 8.5 shows a new Web page for the `MoreHelloApplet` applet.

Listing 8.5. The HTML file for the `MoreHelloApplet` applet.

```
1: <HTML>
2: <HEAD>
3: <TITLE>Hello!</TITLE>
4: </HEAD>
5: <BODY>
6: <P>
7: <APPLET CODE="MoreHelloApplet.class" WIDTH=200 HEIGHT=50>
8: <PARAM NAME=name VALUE="Bonzo">
9: Hello to whoever you are!
10: </APPLET>
11: </BODY>
12: </HTML>
```

Analysis

Note the `<APPLET>` tag, which points to the class file for the applet and has the appropriate width and height (`200` and `50`). Just below it (line 8) is the `<PARAM>` tag, which you use to pass in the value for the name. Here, the `NAME` parameter is simply `name`, and the `VALUE` is the string `"Bonzo"` .

Let's try a second example. Remember that in the code for `MoreHelloApplet` , if no name is specified in a parameter, the default is the name `Laura` . Listing 8.6 creates an HTML file with no parameter tag for `name` .

Listing : Another HTML file for the `MoreHelloApplet` applet.

```
1: <HTML>
2: <HEAD>
3: <TITLE>Hello!</TITLE>
4: </HEAD>
5: <BODY>
6: <P>
7: <APPLET CODE="MoreHelloApplet.class" WIDTH=200 HEIGHT=50>
8: Hello to whoever you are!
9: </APPLET>
10: </BODY>
11: </HTML>
```

Here, because no name was supplied, the applet uses the default, and the result is what you might expect.

Drawing and Filling Objects

The Graphics Class

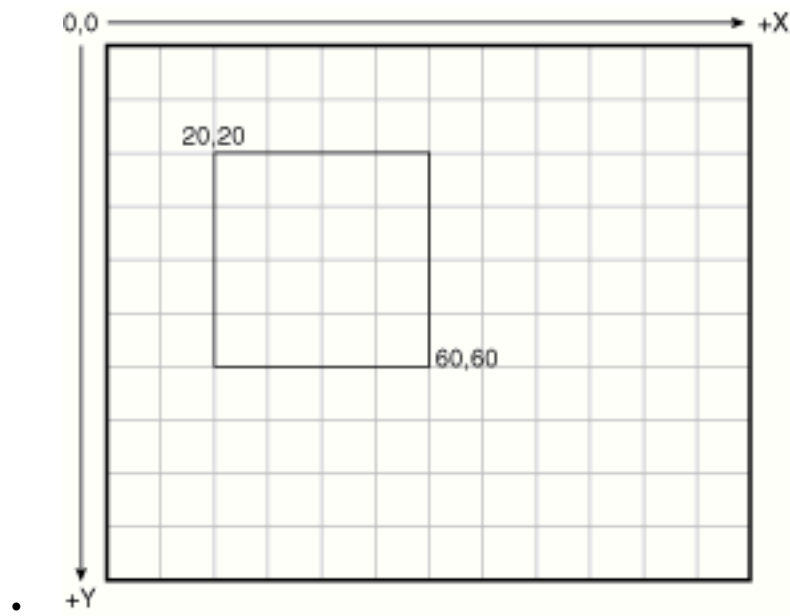
- With the basic graphics capabilities built into Java's class libraries, you can draw *lines, shapes, characters, and images* to the screen inside your applet.
- Most of the graphics operations in Java are methods defined in the `Graphics` class.
- You don't have to create an instance of `Graphics` in order to draw something in your applet; in your applet's `paint()` method (which you learned about yesterday), you are given a `Graphics` object.
- By drawing on that object, you draw onto your applet and the results appear onscreen.
- The `Graphics` class is part of the `java.awt` package, so if your applet does any painting (as it usually will), make sure you import that class at the beginning of your Java file:

```
import java.awt.Graphics;

public class MyClass extends java.applet.Applet {
    ...
}
```

The Graphics Coordinate System

- To draw an object on the screen, you call one of the drawing methods available in the `Graphics` class.
- All the drawing methods have arguments representing endpoints, corners, or starting locations of the object as values in the applet's coordinate system—for example, a line starts at the point `10,10` and ends at the point `20,20`.
- Java's coordinate system has the origin (`0,0`) in the top-left corner. Positive `x` values are to the right and positive `y` values are down. All pixel values are integers; there are no partial or fractional pixels. Figure 9.1 shows how you might draw a simple square by using this coordinate system.
- Java's coordinate system is different from that of many painting and layout programs, which have their `x` and `y` in the bottom left. If you're not used to working with this upside-down graphics system, it may take some practice to get familiar with it.



Drawing and Filling

The `Graphics` class provides a set of simple built-in graphics primitives for drawing, including lines, rectangles, polygons, ovals, and arcs

Lines

- To draw straight lines, use the **`drawLine()`** method. `drawLine()` takes four arguments: the `x` and `y` coordinates of the starting point and the `x` and `y` coordinates of the ending point. So, for example, the following `MyLine` class draws a line from the point `25,25` to the point `75,75`.
- Note that the `drawLine()` method is defined in the `Graphics` class (as are all the other graphics methods you'll learn about today). Here we're using that method for the current graphics context stored in the variable `g`:

```
import java.awt.Graphics;
```

```
public class MyLine extends java.applet.Applet {  
    public void paint(Graphics g)  
    {  
        g.drawLine(25,25,75,75);  
    }  
}
```

Figure below shows how the simple `MyLine` class looks in a Java-enabled browser such as Netscape.



Rectangles

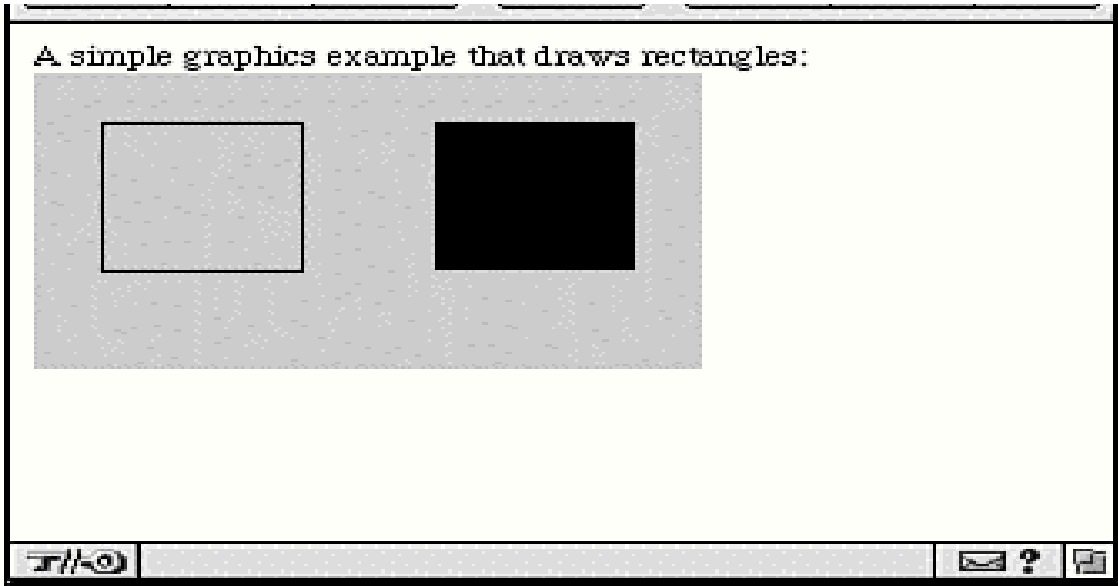
- The Java graphics primitives provide not just one, but three kinds of rectangles:
 - o Plain rectangles
 - o Rounded rectangles, which are rectangles with rounded corners
 - o Three-dimensional rectangles, which are drawn with a shaded border
- For each of these rectangles, you have two methods to choose from: one that draws the rectangle in outline form and one that draws the rectangle filled with color.
- To draw a plain rectangle, use either the `drawRect()` or `fillRect()` methods. Both take ***four arguments:*** the `x` and `y` coordinates of the top-left corner of the rectangle, and the *width and height* of the rectangle to draw. For example, the following class (`MyRect`) draws two squares: The left one is an outline and the right one is filled (Figure 9.3 shows the result):

```
import java.awt.Graphics;
```

```
public class MyRect extends java.applet.Applet {  
    public void paint(Graphics g) {  
        g.drawRect(20,20,60,60);  
        g.fillRect(120,20,60,60);  
    }  
}
```

- Rounded rectangles are, as you might expect, rectangles with rounded corners. The `drawRoundRect()` and `fillRoundRect()` methods to draw rounded rectangles are similar to regular rectangles except that rounded rectangles have two extra arguments for the *width and height of the angle of the corners*.

- Those two arguments determine how far along the edges of the rectangle the arc for the corner will start; the first for the angle along the horizontal plane, the second for the vertical.
- Larger values for the angle width and height make the overall rectangle more rounded; values equal to the width and height of the rectangle itself produce a circle. Figure 9.4 shows some examples of rounded corners.
- The following is a `paint()` method inside a class called `MyRRect` that draws two rounded rectangles: one as an outline with a rounded corner `10` pixels square; the other, filled, with a rounded corner `20` pixels square.



```
import java.awt.Graphics;

public class MyRRect extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawRoundRect(20,20,60,60,10,10);
        g.fillRoundRect(120,20,60,60,20,20);
    }
}
```

- Finally, there are three-dimensional rectangles. These rectangles aren't really 3D; instead, they have a slight shadow effect that makes them appear either raised or indented from the surface of the applet.
- Three-dimensional rectangles have four arguments for the `x` and `y` of the start position and the width and height of the rectangle.
- The fifth argument is a boolean indicating whether the 3D effect is to raise the rectangle (`true`) or indent it (`false`). As with the other rectangles, there are also different methods for drawing and filling: `draw3DRect()` and `fill3DRect()` .
- The following is a class called `My3DRect` , which produces two 3D squares-the left one raised, the right one indented.

```
import java.awt.Graphics;

public class My3DRect extends java.applet.Applet {
public void paint(Graphics g) {
g.draw3DRect(20,20,60,60,true);
g.draw3DRect(120,20,60,60,false);
}
}
```

Polygons

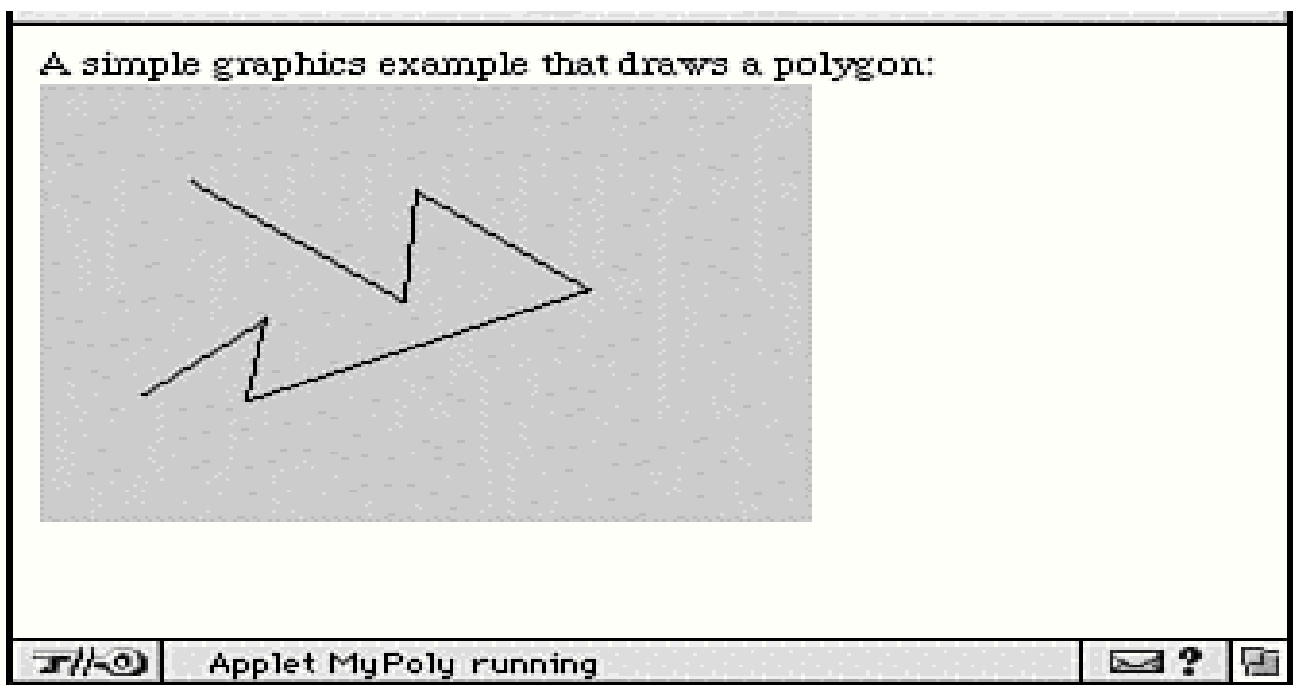
- Polygons are shapes with an unlimited number of sides. To draw a polygon, you need a set of `x` and `y` coordinates. The polygon is then drawn as a set of straight lines from the first point to the second, the second to the third, and so on.
- As with rectangles, you can draw an outline or a filled polygon (using the `drawPolygon()` and `fillPolygon()` methods, respectively). You also have a choice of how you want to indicate the list of coordinates-either as arrays of `x` and `y` coordinates or as an instance of the `Polygon` class.
- Using the first way of drawing polygons, the `drawPolygon()` and `fillPolygon()` methods take three arguments:
 - o An array of integers representing `x` coordinates
 - o An array of integers representing `y` coordinates
 - o An integer for the total number of points

The `x` and `y` arrays should, of course, have the same number of elements.

Here's an example of drawing a polygon's outline using this method.

```
import java.awt.Graphics;

public class MyPoly extends java.applet.Applet {
public void paint(Graphics g) {
int exes[] = { 39,94,97,142,53,58,26 };
int whys[] = { 33,74,36,70,108,80,106 };
int pts = exes.length;
g.drawPolygon(exes,whys,pts);
}
}
```



oNote that Java does not automatically close the polygon; if you want to complete the shape, you have to include the starting point of the polygon at the end of the array. Drawing a filled polygon, however, joins the starting and ending points.

oThe second way of calling `drawPolygon()` and `fillPolygon()` is to use a `Polygon` object to store the individual points of the polygon. The `Polygon` class is useful if you intend to add points to the polygon or if you're building the polygon on-the-fly. Using the `Polygon` class, you can treat the polygon as an object rather than having to deal with individual arrays.

To create a polygon object, you can either first create an empty polygon:

```
Polygon poly = new Polygon();
```

or create a polygon from a set of points using integer arrays, as in the previous example:

```
int exes[] = { 39,94,97,142,53,58,26 };
int whys[] = { 33,74,36,70,108,80,106 };
int pts = exes.length;
Polygon poly = new Polygon(exes,whys,pts);
```

Once you have a polygon object, you can add points to the polygon as you need to:

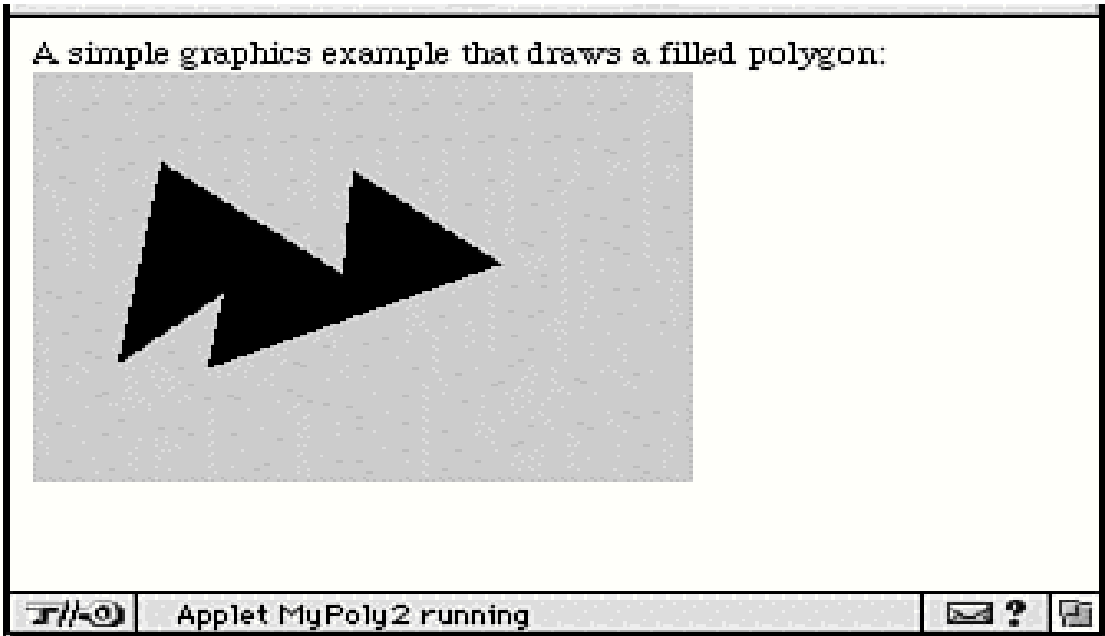
```
poly.addPoint(20,35);
```

Then, to draw the polygon, just use the polygon object as an argument to `drawPolygon()` or `fillPolygon()`. Here's that previous example, rewritten this time with a `Polygon` object. You'll also fill this polygon rather than just drawing its outline.

```
import java.awt.Graphics;

public class MyPoly2 extends java.applet.Applet {

    public void paint(Graphics g) {
        int exes[] = { 39,94,97,142,53,58,26 };
        int whys[] = { 33,74,36,70,108,80,106 };
        int pts = exes.length;
        Polygon poly = new Polygon(exes,whys,pts);
        g.fillPolygon(poly);
    }
}
```



Ovals

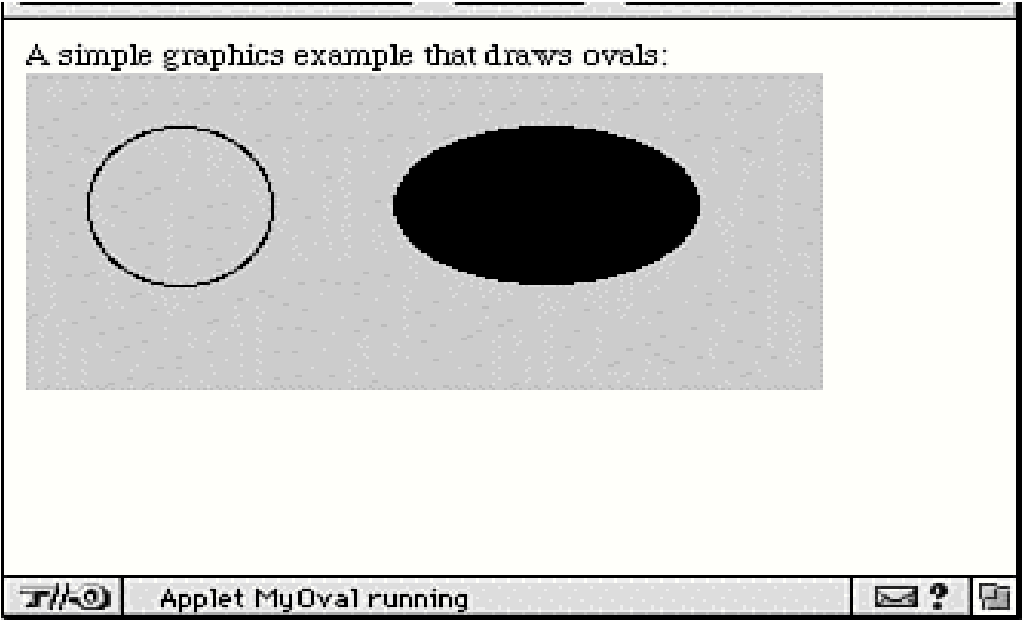
You use ovals to draw ellipses or circles. Ovals are just like rectangles with overly rounded corners. You draw them using four arguments: the `x` and `y` of the top corner, and the width and height of the oval itself. Note that because you're drawing an oval, the starting point is some distance to the left and up from the actual outline of the oval itself. Again, if you think of it as a rectangle, it's easier to place.

As with the other drawing operations, the `drawOval()` method draws an outline of an oval, and the `fillOval()` method draws a filled oval.

The following example draws two ovals—a circle and an ellipse (Figure below shows how these two ovals appear onscreen):

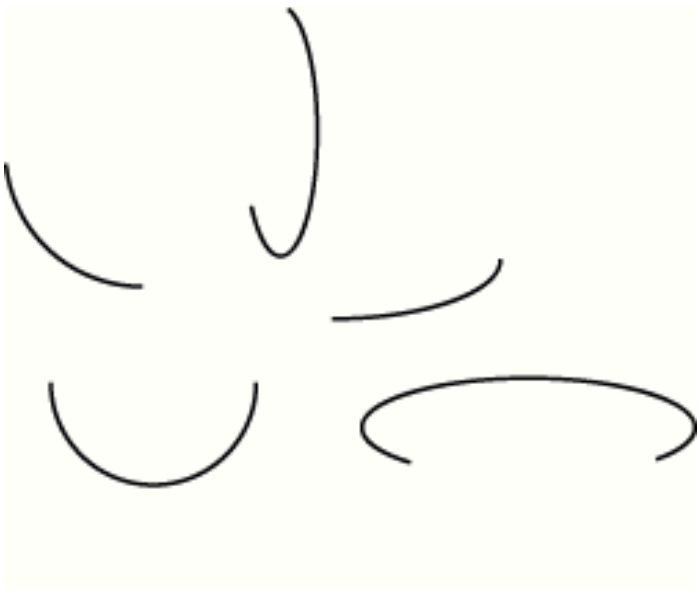
```
import java.awt.Graphics;

public class MyOval extends java.applet.Applet {
public void paint(Graphics g) {
g.drawOval(20,20,60,60);
g.fillOval(120,20,100,60);
}
}
```



Arcs

- Of all the shapes you can construct using methods in the Graphics class, arcs are the most complex to construct, which is why I saved them for last.
- An arc is a part of an oval; in fact, the easiest way to think of an arc is as a section of a complete oval. Figure shows some arcs.



- The `drawArc()` method takes six arguments: the starting corner, the width and height, the angle at which to start the arc, and the degrees to draw it before stopping.
- Once again, there is a `drawArc` method to draw the arc's outline and the `fillArc()` method to fill the arc. Filled arcs are drawn as if they were sections of a pie; instead of joining the two endpoints, both endpoints are joined to the center of the circle.
- The important thing to understand about arcs is that you're actually formulating the arc as an oval and then drawing only some of that.
- The starting corner and width and height are not the starting point and width and height of the actual arc as drawn on the screen; they're the width and height of the full ellipse of which the arc is a part. Those first points determine the size and shape of the arc; the last two arguments (for the degrees) determine the starting and ending points.

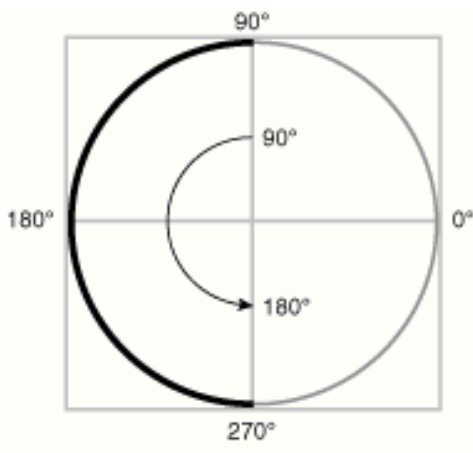
Let's start with a simple arc, a C shape on a circle, as shown in Figure.



To construct the method to draw this arc, the first thing you do is think of it as a complete circle. Then you find the `x` and `y` coordinates and the width and height of that circle. Those four values are the first four arguments to the `drawArc()` or `fillArc()` methods. Figure 9.12 shows how to get those values from the arc.

To get the last two arguments, think in degrees around the circle, going counterclockwise. Zero degrees is at 3 o'clock, 90 degrees is at 12 o'clock, 180 at 9 o'clock, and 270 at 6 o'clock. The start of the arc is the degree value of the start of the arc. In this example, the starting point is the top of the C at 90 degrees; 90 is the fifth argument.

The sixth and last argument is another degree value indicating how far around the circle to sweep and the direction to go in (it's not the ending degree angle, as you might think). In this case, because you're going halfway around the circle, you're sweeping 180 degrees-and 180 is therefore the last argument in the arc. The important part is that you're sweeping 180 degrees counterclockwise, which is in the positive direction in Java. If you are drawing a backwards C, you sweep 180 degrees in the negative direction, and the last argument is `-180`. See Figure for the final illustration of how this works.



Note

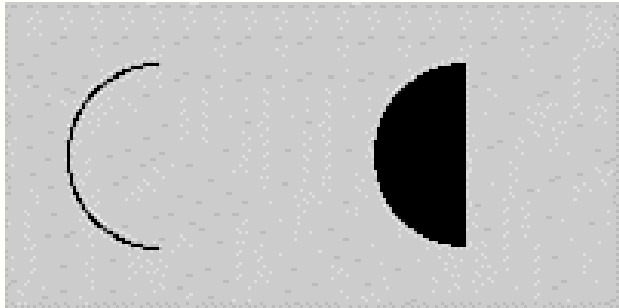
It doesn't matter which side of the arc you start with. Because the shape of the arc has already been determined by the complete oval it's a section of, starting at either endpoint will work.

Here's the code for this example; you'll draw an outline of the C and a filled C to its right, as shown in Figure.

```
import java.awt.Graphics;

public class MyOval extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawArc(20,20,60,60,90,180);
        g.fillArc(120,20,60,60,90,180);
    }
}
```

A simple graphics example that draws circular arcs:



Circles are an easy way to visualize arcs on circles; arcs on ellipses are slightly more difficult.

Like the arc on the circle, this arc is a piece of a complete oval, in this case, an elliptical oval. By completing the oval that this arc is a part of, you can get the starting points and the width and height arguments for the `drawArc()` or `fillArc()` method

Then all you need is to figure out the starting angle and the angle to sweep. This arc doesn't start on a nice boundary such as 90 or 180 degrees, so you'll need some trial and error. This arc starts somewhere around 25 degrees, and then sweeps clockwise about 130 degrees .

With all portions of the arc in place, you can write the code. Here's the Java code for this arc, both drawn and filled (note in the filled case how filled arcs are drawn as if they were pie sections):

```
import java.awt.Graphics;

public class MyOval extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawArc(10,20,150,50,25,-130);
        g.fillArc(10,80,150,50,25,-130);
    }
}
```

To summarize, here are the steps to take to construct arcs in Java:

1. Think of the arc as a slice of a complete oval.
2. Construct the full oval with the starting point and the width and height (it often helps to draw the full oval on the screen to get an idea of the right positioning).
3. Determine the starting angle for the beginning of the arc.
4. Determine how far to sweep the arc and in which direction (counterclockwise indicates positive values, clockwise indicates negative).

A Simple Graphics Example

Here's an example of an applet that uses many of the built-in graphics primitives to draw a rudimentary shape. In this case, it's a lamp with a spotted shade (or a sort of cubist mushroom, depending on your point of view). Listing 9.1 has the complete code for the lamp; Figure below shows the resulting applet.

Listing : The `Lamp` class.

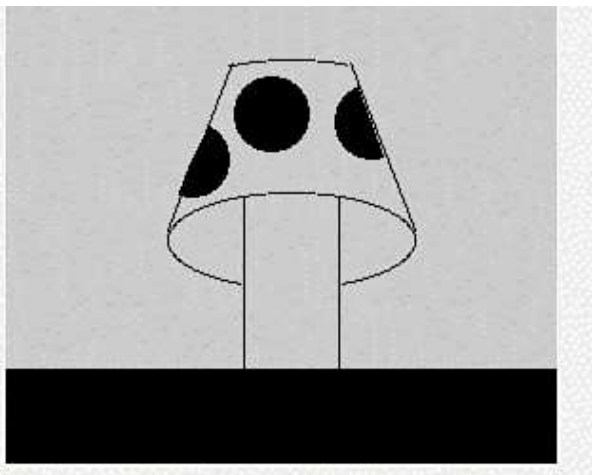
```
1: import java.awt.*;
2:
3: public class Lamp extends java.applet.Applet {
```



```

4:
5:  public void paint(Graphics g) {
6:      // the lamp platform
7:      g.fillRect(0,250,290,290);
8:
9:      // the base of the lamp
10:     g.drawLine(125,250,125,160);
11:     g.drawLine(175,250,175,160);
12:
13:     // the lamp shade, top and bottom edges
14:     g.drawArc(85,157,130,50,-65,312);
15:     g.drawArc(85,87,130,50,62,58);
16:
17:     // lamp shade, sides
18:     g.drawLine(85,177,119,89);
19:     g.drawLine(215,177,181,89);
20:
21:     // dots on the shade
22:     g.fillArc(78,120,40,40,63,-174);
23:     g.fillOval(120,96,40,40);
24:     g.fillArc(173,100,40,40,110,180);
25: }
26: }

```



Text and Fonts

- Using the `Graphics` class, you can also print text on the screen, in conjunction with the `Font` class (and, sometimes, the `FontMetrics` class). The `Font` class represents a given font-its name, style, and point size-and `FontMetrics` gives you information about that font (for example, the actual height or width of a given character) so that you can precisely lay out text in your applet.

- Note that the text here is drawn to the screen once and intended to stay there. You'll learn about entering text from the keyboard later this week.

Creating Font Objects

- To draw text to the screen, first you need to create an instance of the `Font` class. Font objects represent an individual font-that is, its name, style (bold, italic), and point size. Font names are strings representing the family of the font, for example, `"TimesRoman"` , `"Courier"` , or `"Helvetica"` . Font styles are constants defined by the `Font` class; you can get to them using class variables-for example, `Font.PLAIN` , `Font.BOLD` , or `Font.ITALIC` . Finally, the point size is the size of the font, as defined by the font itself; the point size may or may not be the height of the characters.
- To create an individual font object, use these three arguments to the `Font` class's `new` constructor:

```
Font f = new Font("TimesRoman", Font.BOLD, 24);
```

- This example creates a font object for the `TimesRoman BOLD` font, in `24` points. Note that like most Java classes, you have to import the `java.awt.Font` class before you can use it.

Tip

Font styles are actually integer constants that can be added to create combined styles; for example, `Font.BOLD + Font.ITALIC` produces a font that is both bold and italic.

- The fonts you have available to you in your applets depend on which fonts are installed on the system where the applet is running. If you pick a font for your applet and that font isn't available on the current system, Java will substitute a default font (usually Courier). You can get an array of the names of the current fonts available in the system using this bit of code:

```
String[] fontlist = this.getToolkit().getFontList();
```

- From this list, you can then often intelligently decide which fonts you want to use in your applet.

For best results, however, it's a good idea to stick with standard fonts such as `"TimesRoman"` , `"Helvetica"` , and `"Courier"` .

- With a font object in hand, you can draw text on the screen using the methods `drawChars()` and `drawString()`. First, though, you need to set the current font to your font object using the `setFont()` method.
- The current font is part of the graphics state that is kept track of by the `Graphics` object on which you're drawing. Each time you draw a character or a string to the screen, Java draws that text in the current font. To change the font of the text, therefore, first change the current font. The following `paint()` method creates a new font, sets the current font to that font, and draws the string "This is a big font." , at the point 10,100 :

```
public void paint(Graphics g) {
    Font f = new Font("TimesRoman", Font.PLAIN, 72);
    g.setFont(f);
    g.drawString("This is a big font.", 10, 100);
}
```

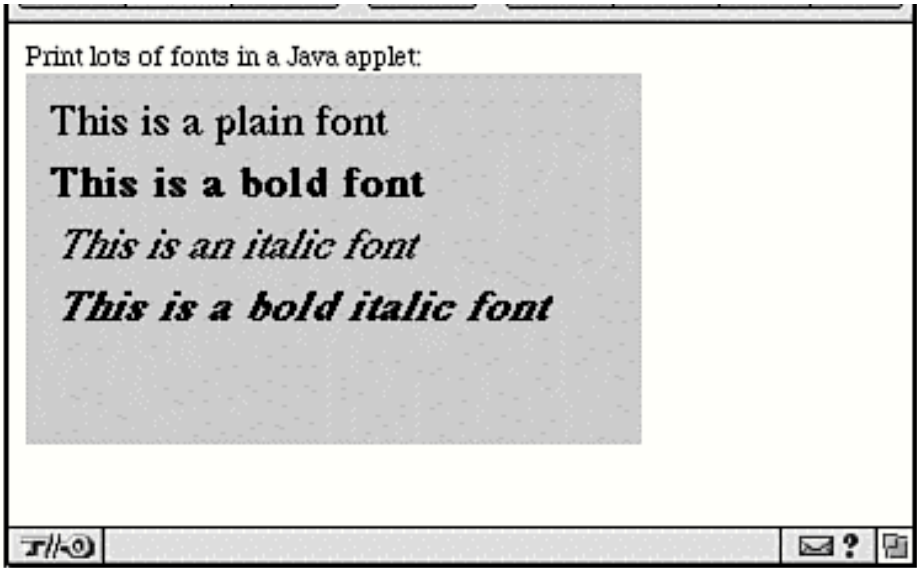
- The latter two arguments to `drawString()` determine the point where the string will start. The `x` value is the start of the leftmost edge of the text; `y` is the baseline for the entire string.
- Similar to `drawString()` is the `drawChars()` method that, instead of taking a string as an argument, takes an array of characters. `drawChars()` has five arguments: the array of characters, an integer representing the first character in the array to draw, another integer for the last character in the array to draw (all characters between the first and last are drawn), and the `x` and `y` for the starting point. Most of the time, `drawString()` is more useful than `drawChars()`.

Listing below shows an applet that draws several lines of text in different fonts; Figure 9.20 shows the result.

Listing : Many different fonts.

```
1: import java.awt.Font;
2: import java.awt.Graphics;
3:
4: public class ManyFonts extends java.applet.Applet {
5:
6:     public void paint(Graphics g) {
7:         Font f = new Font("TimesRoman", Font.PLAIN, 18);
8:         Font fb = new Font("TimesRoman", Font.BOLD, 18);
9:         Font fi = new Font("TimesRoman", Font.ITALIC, 18);
10:        Font fbi = new Font("TimesRoman", Font.BOLD + Font.ITALIC, 18);
11:
```

```
12:     g.setFont(f);
13:     g.drawString("This is a plain font", 10, 25);
14:     g.setFont(fb);
15:     g.drawString("This is a bold font", 10, 50);
16:     g.setFont(fi);
17:     g.drawString("This is an italic font", 10, 75);
18:     g.setFont(fbi);
19:     g.drawString("This is a bold italic font", 10, 100);
20: }
21:
22: }
```



Finding Out Information About a Font

Sometimes you may want to make decisions in your Java program based on the qualities of the current font—for example, its point size and the total height of its characters. You can find out some basic information about fonts and font objects by using simple methods on `Graphics` and on the `Font` objects. Table 9.1 shows some of these methods.

Table ; Font methods.

<i>Method Name</i>	<i>In Object</i>	<i>Action</i>
<code>getFont()</code>	<code>Graphics</code>	Returns the current font object as previously set by <code>setFont()</code>
<code>getName()</code>	<code>Font</code>	Returns the name of the font as a string
<code>getSize()</code>	<code>Font</code>	Returns the current font size (an integer)
<code>getStyle()</code>	<code>Font</code>	Returns the current style of the font (styles are integer constants: 0 is plain, 1 is bold, 2 is italic, 3 is bold italic)

isPlain()	Font	Returns true or false if the font's style is plain
isBold()	Font	Returns true or false if the font's style is bold
isItalic()	Font	Returns true or false if the font's style is italic

For more detailed information about the qualities of the current font (for example, the length or height of given characters), you need to work with font metrics. The `FontMetrics` class describes information specific to a given font: the leading between lines, the height and width of each character, and so on. To work with these sorts of values, you create a `FontMetrics` object based on the current font by using the applet method `getFontMetrics()` :

```
Font f = new Font("TimesRoman", Font.BOLD, 36);
FontMetrics fmetrics = getFontMetrics(f);
g.setFont(f);
```

Table 9.2 shows some of the things you can find out using font metrics. All these methods should be called on a `FontMetrics` object.

Table 9.2. Font metrics methods.

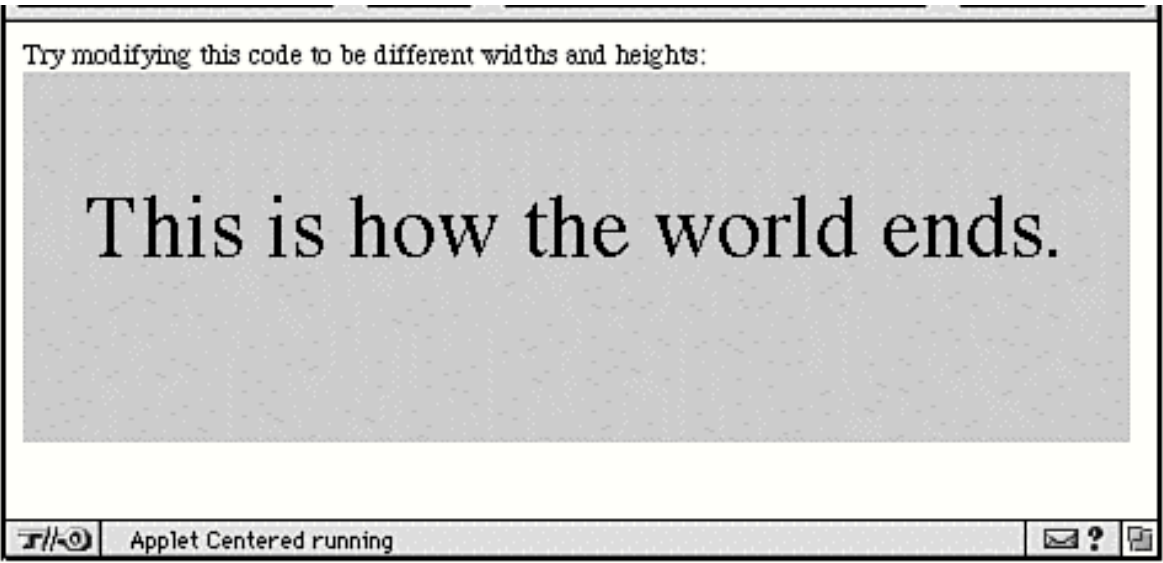
<i>Method Name</i>	<i>Action</i>
<code>stringWidth(string)</code>	Given a string, returns the full width of that string, in pixels
<code>charWidth(char)</code>	Given a character, returns the width of that character
<code>getAscent()</code>	Returns the ascent of the font, that is, the distance between the font's baseline and the top of the characters
<code>getDescent()</code>	Returns the descent of the font-that is, the distance between the font's baseline and the bottoms of the characters (for characters such as p and q that drop below the baseline)
<code>getLeading()</code>	Returns the leading for the font, that is, the spacing between the descent of one line and the ascent of another line
<code>getHeight()</code>	Returns the total height of the font, which is the sum of the ascent, descent, and leading value

As an example of the sorts of information you can use with font metrics, Listing 9.3 shows the Java code for an applet that automatically centers a string horizontally and vertically inside an applet. The centering position is different depending on the font and font size; by using font metrics to find out the actual size of a string, you can draw the string in the appropriate place.

Figure shows the result (which is less interesting than if you actually compile and experiment with various applet and font sizes).

Listing : Centering a string.

```
1: import java.awt.Font;
2: import java.awt.Graphics;
3: import java.awt.FontMetrics;
4:
5: public class Centered extends java.applet.Applet {
6:
7:     public void paint(Graphics g) {
8:         Font f = new Font("TimesRoman", Font.PLAIN, 36);
9:         FontMetrics fm = getFontMetrics(f);
10:        g.setFont(f);
11:
12:        String s = "This is how the world ends.";
13:        int xstart = (size().width - fm.stringWidth(s)) / 2;
14:        int ystart = size().height / 2;
15:
16:        g.drawString(s, xstart, ystart);
17:    }
18:}
```



Analysis

Note the `size()` method in lines 13 and 14, which returns the width and height of the overall applet area as a `Dimension` object. You can then get to the individual width and height using the `width` and `height` instance variables of that `Dimension`, here by chaining the method call and the

variable name. Getting the current applet size in this way is a better idea than hard coding the size of the applet into your code; this code works equally well with an applet of any size.

Note also that the line of text, as shown in Figure isn't precisely vertically centered in the applet bounding box. This example centers the baseline of the text inside the applet; using the `getAscent()` and `getDescent()` methods from the `FontMetrics` class (to get the number of pixels from the baseline to the top of the characters and the number of pixels from the baseline to the bottom of the characters), you can figure out exactly the middle of the line of text.

Color

- Drawing black lines and text on a gray background is all very nice, but being able to use different colors is much nicer. Java provides methods and behaviors for dealing with color in general through the `Color` class, and also provides methods for setting the current foreground and background colors so that you can draw with the colors you created.
- Java's abstract color model uses 24-bit color, wherein a color is represented as a combination of red, green, and blue values. Each component of the color can have a number between 0 and 255 . 0,0,0 is black, 255,255,255 is white, and Java can represent millions of colors between as well.
- Java's abstract color model maps onto the color model of the platform Java is running on, which usually has only 256 or fewer colors from which to choose. If a requested color in a color object is not available for display, the resulting color may be mapped to another or dithered, depending on how the browser viewing the color implemented it, and depending on the platform on which you're running. In other words, although Java gives the capability of managing millions of colors, very few may actually be available to you in real life.

Using Color Objects

To draw an object in a particular color, you must create an instance of the `Color` class to represent that color. The `Color` class defines a set of standard color objects, stored in class variables, to quickly get a color object for some of the more popular colors. For example, `Color.red` returns a `Color` object representing red (RGB values of 255 ,0 , and 0), `Color.white` returns a white color (RGB values of 255 , 255 , and 255), and so on. Table below shows the standard colors defined by variables in the `Color` class.

Table Standard colors.

<i>Color Name</i>	<i>RGB Value</i>
Color.white	255,255,255
Color.black	0,0,0
Color.lightGray	192,192,192
Color.gray	128,128,128
Color.darkGray	64,64,64
Color.red	255,0,0
Color.green	0,255,0
Color.blue	0,0,255
Color.yellow	255,255,0
Color.magenta	255,0,255
Color.cyan	0,255,255
Color.pink	255,175,175
Color.orange	255,200,0

If the color you want to draw in is not one of the standard `Color` objects, fear not. You can create a color object for any combination of red, green, and blue, as long as you have the values of the color you want. Just create a new color object:

```
Color c = new Color(140,140,140);
```

This line of Java code creates a color object representing a dark gray. You can use any combination of red, green, and blue values to construct a color object.

Alternatively, you can create a color object using three floats from `0.0` to `1.0` :

```
Color c = new Color(0.55,0.55,0.55);
```


Testing and Setting the Current Colors

To draw an object or text using a color object, you have to set the current color to be that color object, just as you have to set the current font to the font in which you want to draw. Use the `setColor()` method (a method for `Graphics` objects) to do this:

```
g.setColor(Color.green);
```

After you set the current color, all drawing operations will occur in that color.

In addition to setting the current color for the graphics context, you can also set the background and foreground colors for the applet itself by using the `setBackground()` and `setForeground()` methods. Both of these methods are defined in the `java.awt.Component` class, which `Applet` -and therefore your classes-automatically inherits.

The `setBackground()` method sets the background color of the applet, which is usually a light gray (to match the default background of the browser). It takes a single argument, a `Color` object:

```
setBackground(Color.white);
```

The `setForeground()` method also takes a single color as an argument, and it affects everything that has been drawn on the applet, regardless of the color in which it has been drawn. You can use `setForeground()` to change the color of everything in the applet at once, rather than having to redraw everything:

```
setForeground(Color.black);
```

In addition to the `setColor()`, `setForeground()`, and `setBackground()` methods, there are corresponding `get` methods that enable you to retrieve the current graphics color, background, or foreground. Those methods are `getColor()` (defined in `Graphics` objects), `getForeground()` (defined in `Applet`), and `getBackground()` (also in `Applet`). You can use these methods to choose colors based on existing colors in the applet:

```
setForeground(g.getColor());
```

A Simple Color Example

Listing below shows the code for an applet that fills the applet's drawing area with square boxes, each of which has a randomly chosen color in it. It's written so that it can handle any size of applet and automatically fill the area with the right number of boxes.

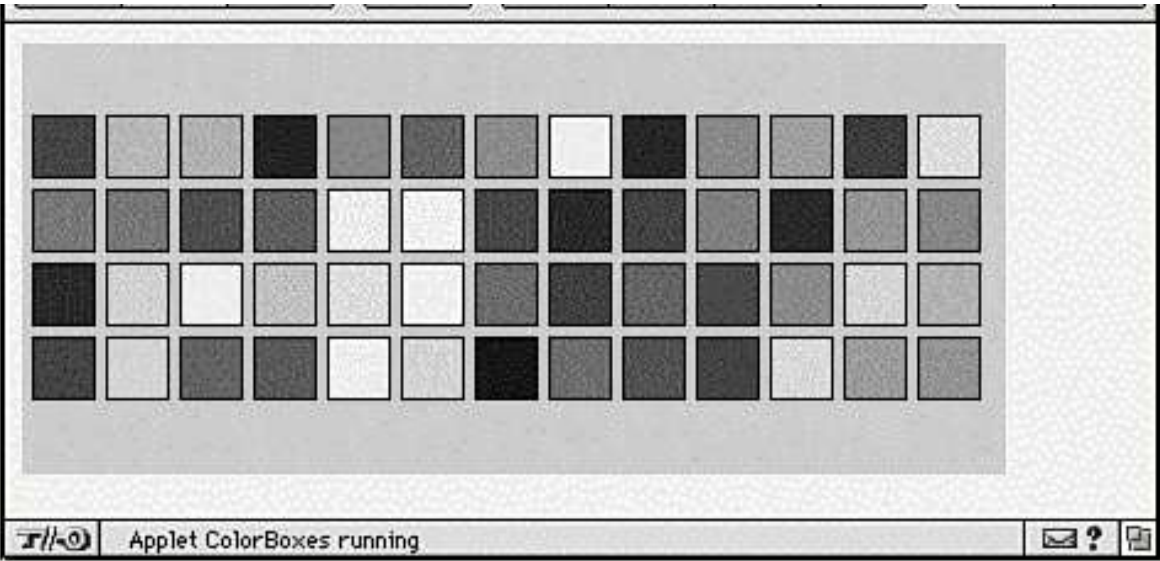
Listing Random color boxes.

```
1: import java.awt.Graphics;
2: import java.awt.Color;
3:
4: public class ColorBoxes extends java.applet.Applet {
5:
6:     public void paint(Graphics g) {
7:         int rval, gval, bval;
8:
9:         for (int j = 30; j < (size().height - 25); j += 30)
10:             for (int i = 5; i < (size().width - 25); i += 30) {
11:                 rval = (int)Math.floor(Math.random() * 256);
12:                 gval = (int)Math.floor(Math.random() * 256);
13:                 bval = (int)Math.floor(Math.random() * 256);
14:
15:                 g.setColor(new Color(rval,gval,bval));
16:                 g.fillRect(i, j, 25, 25);
17:                 g.setColor(Color.black);
18:                 g.drawRect(i-1, j-1, 25, 25);
19:             }
20:     }
21: }
```

Analysis

The two `for` loops are the heart of this example; the first one draws the rows, and the second draws the individual boxes within each row. When a box is drawn, the random color is calculated first, and then the box is drawn. A black outline is drawn around each box, because some of them tend to blend into the background of the applet.

Because this `paint` method generates new colors each time the applet is painted, you can regenerate the colors by moving the window around or by covering the applet's window with another one (or by reloading the page). Figure below shows the final applet (although given that this picture is black and white, you can't get the full effect of the multicolored squares).



Creating User Interfaces with the awt

An awt Overview

The awt provides the following:

A full set of user interface (UI) widgets and other components, including windows, menus, buttons, check boxes, text fields, scrollbars, and scrolling lists

Support for UI containers, which can contain other embedded containers or UI widgets

An event system for managing system and user events among parts of the awt

Mechanisms for laying out components in a way that enables platform-independent UI design

- The basic idea behind the awt is that a graphical Java program is a set of nested components, starting from the outermost window all the way down to the smallest UI component.
- Components can include things you can actually see on the screen, such as windows, menu bars, buttons, and text fields, and they can also include containers, which in turn can contain other components
- This nesting of components within containers within other components creates a hierarchy of components, from the smallest check box inside an applet to the overall window on the screen.
- The hierarchy of components determines the arrangement of items on the screen and inside other items, the order in which they are painted, and how events are passed from one component to another.
- These are the major components you can work with in the awt:

Containers. Containers are generic awt components that can contain other components, including other containers. The most common form of container is the *panel*, which represents a container that can be displayed onscreen. Applets are a form of panel (in fact, the `Applet` class is a subclass of the `Panel` class).

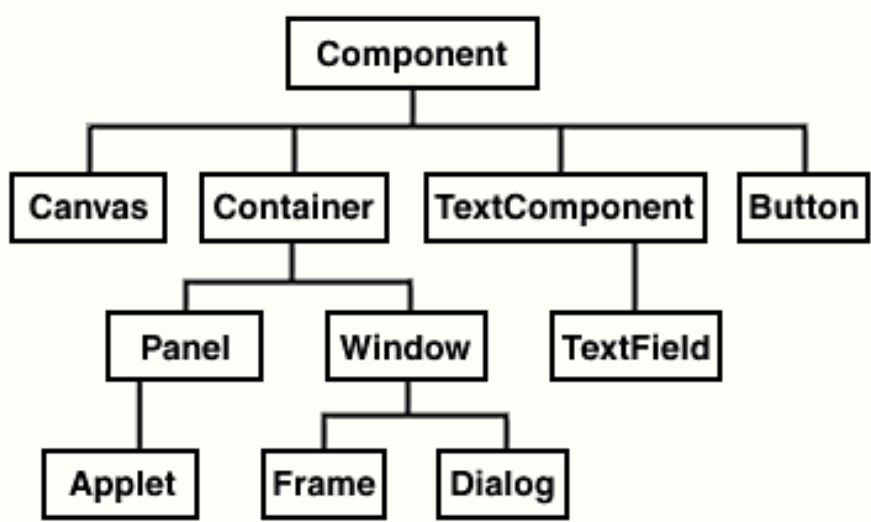
Canvases. A canvas is a simple drawing surface. Although you can draw on panels (as you've been doing all along), canvases are good for painting images or performing other graphics operations.

UI components. These can include buttons, lists, simple pop-up menus, check boxes, text fields, and other typical elements of a user interface.

Window construction components. These include windows, frames, menu bars, and dialog boxes. They are listed separately from the other UI components because you'll use these less often-

particularly in applets. In applets, the browser provides the main window and menu bar, so you don't have to use these. Your applet may create a new window, however, or you may want to write your own Java application that uses these components. (You'll learn about these tomorrow.)

- The classes inside the `java.awt` package are written and organized to mirror the abstract structure of containers, components, and individual UI components.
- Figure below shows some of the class hierarchy that makes up the main classes in the awt.



- The root of most of the awt components is the class `Component` , which provides basic display and event-handling features.
- The classes `Container` , `Canvas` , `TextComponent` , and many of the other UI components inherit from `Component` .
- Inheriting from the `Container` class are objects that can contain other awt components-the `Panel` and `Window` classes, in particular.
- Note that the `java.applet.Applet` class, even though it lives in its own package, inherits from `Panel` , so your applets are an integral part of the hierarchy of components in the awt system.

- A graphical user interface-based application that you write by using the awt can be as complex as you like, with dozens of nested containers and components inside each other.
- The awt was designed so that each component can play its part in the overall awt system without needing to duplicate or keep track of the behavior of other parts in the system.
- **Layout Managers**: In addition to the components themselves, the awt also includes a set of layout managers. Layout managers determine how the various components are arranged when they are

displayed onscreen, and their various sizes relative to each other. Because Java applets and applications that use the awt can run on different systems with different displays, different fonts, and different resolutions, you cannot just stick a particular component at a particular spot on the window. Layout managers help you create UI layouts that are dynamically arranged and can be displayed anywhere the applet or application might be run.

The Basic User Interface Components

oThe simplest form of awt component is the basic UI component.

Basic UI components: *labels, buttons, check boxes, choice menus, and text fields* .

Procedure for creating the component

oFirst create the component and

oThen add it to the panel that holds it, at which point it is displayed on the screen.

oTo add a component to a panel (such as your applet, for example), use the

add() **method:**

```
public void init() {  
    Button b = new Button("OK");  
    add(b);  
}
```

OR

Create the component and add to the panel at the same time;

```
add(new Button("OK"));
```

Labels

oThe simplest form of UI component is the label, which is, effectively, a text string that you can use to label other UI components.

oLabels are not editable; they just label other components on the screen.

A label is an uneditable text string that acts as a description for other awt components.

To create a label, use one of the following constructors:

- **Label()** creates an empty label, with its text aligned left.
- **Label(*String*)** creates a label with the given text string, also aligned left.
- **Label(*String* , *int*)** creates a label with the given text string and the given alignment. The available alignment numbers are stored in class variables in `Label` , making them easier to remember:
`Label.RIGHT` , `Label.LEFT` , and `Label.CENTER` .

```
import java.awt.*;

public class LabelTest extends java.applet.Applet {

    public void init() {
        setFont(new Font ("Helvetica", Font.BOLD, 14));
        setLayout(new GridLayout(3,1));
        add(new Label("aligned left", Label.LEFT));
        add(new Label("aligned center", Label.CENTER));
        add(new Label("aligned right", Label.RIGHT));
    }
}
```



Buttons

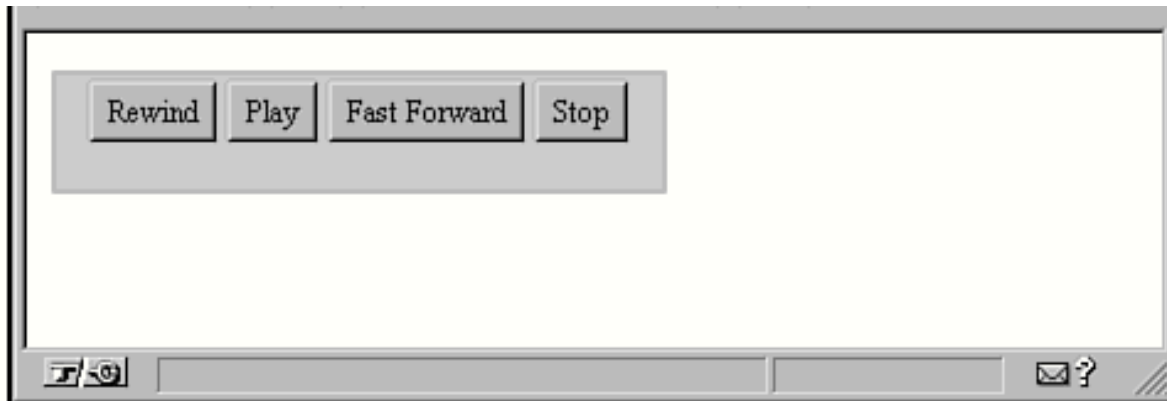
- o Buttons are simple UI components that trigger some action in your interface when they are pressed.
- o For example, a calculator applet might have buttons for each number and operator, or a dialog box might have buttons for OK and Cancel.
- o A *button* is a UI component that, when "pressed" (selected) with the mouse, triggers some action.

To create a button, use one of the following constructors:

- **Button()** creates an empty button with no label.
- **Button(String)** creates a button with the given string as a label.

```
public class ButtonTest extends java.applet.Applet {

public void init() {
add(new Button("Rewind"));
add(new Button("Play"));
add(new Button("Fast Forward"));
add(new Button("Stop"));
}
}
```



Check Boxes

- o *Check boxes* are user-interface components that have two states: on and off (or checked and unchecked, selected and unselected, true and false, and so on).
- o Unlike buttons, check boxes usually don't trigger direct actions in a UI, but instead are used to indicate optional features of some other action.

Check boxes can be used in two ways:

- Nonexclusive: Given a series of check boxes, any of them can be selected.
- Exclusive: Given a series, only one check box can be selected at a time.

The latter kind of check boxes are called radio buttons or check box groups, and are described in the next section.

Check boxes are UI components that can be selected or deselected (checked or unchecked) to provide options. Nonexclusive check boxes can be checked or unchecked independently of other check boxes. Exclusive check boxes, sometimes called *radio buttons*, exist in groups; only one in the group can be checked at one time.

Nonexclusive check boxes can be created by using the `Checkbox` class. You can create a check box using one of the following constructors:

- `Checkbox()` creates an empty check box, unselected.
- `Checkbox(String)` creates a check box with the given string as a label.

- **Checkbox**(*String* , *null* , *boolean*) creates a check box that is either selected or deselected based on whether the boolean argument is `true` or `false` , respectively. (The `null` is used as a placeholder for a group argument. Only radio buttons have groups, as you'll learn in the next section.)

Figure 13.5 shows a few simple check boxes (only `Underwear` is selected) generated using the following code:

```
import java.awt.*;

public class CheckboxTest extends java.applet.Applet {

    public void init() {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        add(new Checkbox("Shoes"));
        add(new Checkbox("Socks"));
        add(new Checkbox("Pants"));
        add(new Checkbox("Underwear", null, true));
        add(new Checkbox("Shirt"));
    }

}
```



Radio Buttons

- o Radio buttons have the same appearance as check boxes, but only one in a series can be selected at a time.
- o To create a series of radio buttons, first create an instance of **CheckboxGroup** :

```
CheckboxGroup cbg = new CheckboxGroup();
```

- o Then create and add the individual check boxes using the constructor with **three arguments** the first is the label, second is the group, and the third is whether that check box is selected).

- oNote that because radio buttons, by definition, have only one in the group selected at a time, the last `true` to be added will be the one selected by default:

```
add(new Checkbox("Yes", cbg, true);
add(new Checkbox("No", cbg, false);
```

Here's a simple example (the results of which are shown in Figure 13.6):

```
import java.awt.*;

public class CheckboxGroupTest extends java.applet.Applet {

    public void init() {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        CheckboxGroup cbg = new CheckboxGroup();

        add(new Checkbox("Red", cbg, false));
        add(new Checkbox("Blue", cbg, false));
        add(new Checkbox("Yellow", cbg, false));
        add(new Checkbox("Green", cbg, true));
        add(new Checkbox("Orange", cbg, false));
        add(new Checkbox("Purple", cbg, false));
    }
}
```



Choice Menus

- oChoice menus are pop-up (or pull-down) menus from which you can select an item.
- oThe menu then displays that choice on the screen. The function of a choice menu is the same across platforms, but its actual appearance may vary from platform to platform.
- oNote that choice menus can have ***only one item selected at a time***.
- oIf you want to be able to choose multiple items from the menu, use a scrolling list instead.
- o *Choice menus* are pop-up menus of items from which you can choose one item.

- o To create a choice menu,
 - o create an *instance of the Choice* class and then
 - o use the *addItem()* *method* to add individual items to it in the order in which they should appear.
 - o Finally, *add the entire choice menu to the panel in the usual way* .

```
import java.awt.*;

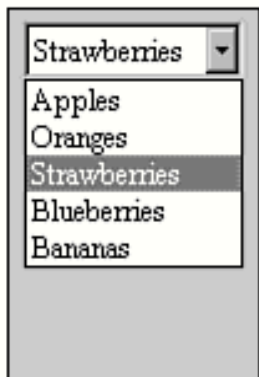
public class ChoiceTest extends java.applet.Applet {

    public void init() {
        Choice c = new Choice();

        c.addItem("Apples");
        c.addItem("Oranges");
        c.addItem("Strawberries");
        c.addItem("Blueberries");
        c.addItem("Bananas");

        add(c);
    }
}
```

- o Even after your choice menu has been added to a panel, you can continue to add items to that menu with the *addItem()* method



Text Fields

- o Text fields allow you to enter and edit text.
- o Text fields are generally only a single line and do not have scrollbars;
- o Text areas, are better for larger amounts of text.
- o Text fields are different from labels in that they can be edited; labels are good for just displaying text, text fields for getting text input from the user.

Text fields provide an area where you can enter and edit a single line of text.

To create a text field, use one of the following constructors:

- ***TextField()*** creates an empty `TextField` that is 0 characters wide (it will be resized by the current layout manager).
- ***TextField(int)*** creates an empty text field. The integer argument indicates the minimum number of characters to display.
- ***TextField(String)*** creates a text field initialized with the given string. The field will be automatically resized by the current layout manager.
- ***TextField(String, int)*** creates a text field some number of characters wide (the integer argument) containing the given string. If the string is longer than the width, you can select and drag portions of the text within the field, and the box will scroll left or right.

For example, the following line creates a text field 30 characters wide with the string "Enter Your Name" as its initial contents:

```
TextField tf = new TextField("Enter Your Name", 30);  
add(tf);
```

Tip

Text fields include only the editable field itself. You usually need to include a label with a text field to indicate what belongs in that text field.

You can also create a text field that obscures the characters typed into it—for example, for password fields.

To do this, first create the text field itself; then use the `setEchoCharacter()` method to set the character that is echoed on the screen. Here is an example:

```
TextField tf = new TextField(30);  
tf.setEchoCharacter('*');
```

Figure 13.8 shows three text boxes (and labels) that were created using the following code:

```
add(new Label("Enter your Name"));  
add(new TextField("your name here", 45));  
add(new Label("Enter your phone number"));  
add(new TextField(12));  
add(new Label("Enter your password"));  
TextField t = new TextField(20);  
t.setEchoCharacter('*');  
add(t);
```

Enter your name:	<input type="text" value="your name here"/>
Enter your phone number:	<input type="text" value="434 235 2354"/>
Enter your password:	<input type="password" value="*****"/>

The text in the first field (`your name here`) was initialized in the code; I typed the text in the remaining two boxes just before taking the snapshot.

Text fields inherit from the class `TextComponent` and have a whole suite of methods, both inherited from that class and defined in their own class, that may be useful to you in your Java programs..

Panels and Layout

- awt panels can contain UI components or other panels. The question now is how those components are actually arranged and displayed onscreen.
- In other windowing systems, UI components are often arranged using hard-coded pixel measurements-put a text field at the position `10,30` , for example-the same way you used the graphics operations to paint squares and ovals on the screen.
- In the awt, your UI design may be displayed on many different window systems on many different screens and with many different kinds of fonts with different font metrics. Therefore, you need a more flexible method of arranging components on the screen so that a layout that looks nice on one platform isn't a jumbled, unusable mess on another.
- For just this purpose, Java has layout managers, insets, and hints that each component can provide to help dynamically lay out the screen.

Layout Managers: An Overview

- The actual appearance of the awt components on the screen is usually determined by two things:
 - o how those components are added to the panel that holds them (either the order or through arguments to `add()`) and

o the layout manager that panel is currently using to lay out the screen. The layout manager determines how portions of the screen will be sectioned and how components within that panel will be placed.

- The *layout manager* determines how awt components are dynamically arranged on the screen.
- Each panel on the screen can have its own layout manager. By nesting panels within panels, and using the appropriate layout manager for each one, you can often arrange your UI to group and arrange components in a way that is functionally useful and that looks good on a variety of platforms and windowing systems.

The awt provides five basic layout managers:

- o `FlowLayout` ,
- o `GridLayout` ,
- o `BorderLayout` ,
- o `CardLayout` , and
- o `GridBagLayout` .

- To create a layout manager for a given panel, create an instance of that layout manager and then use the `setLayout()` method for that panel.
- This example sets the layout manager of the entire enclosing applet panel:

```
public void init() {  
    setLayout(new FlowLayout());  
}
```

- Setting the default layout manager, like creating user-interface components, is best done during the applet's initialization, which is why it's included here.
- After the layout manager is set, you can start adding components to the panel.
- The order in which components are added or the arguments you use to add those components is often significant, depending on which layout manager is currently active.

The following sections describe the five basic Java awt layout managers.

The `FlowLayout` Class

- The `FlowLayout` class is the most basic of layouts.

- Using flow layout, components are added to the panel one at a time, row by row. If a component doesn't fit onto a row, it's wrapped onto the next row.
- The flow layout also has an alignment, which determines the alignment of each row. By default, each row is centered.
- *Flow layout* arranges components from left to right in rows. The rows are aligned left, right, or centered.
- To create a basic flow layout with a centered alignment, use the following line of code in your panel's initialization (because this is the default pane layout, you don't need to include this line if that is your intent):

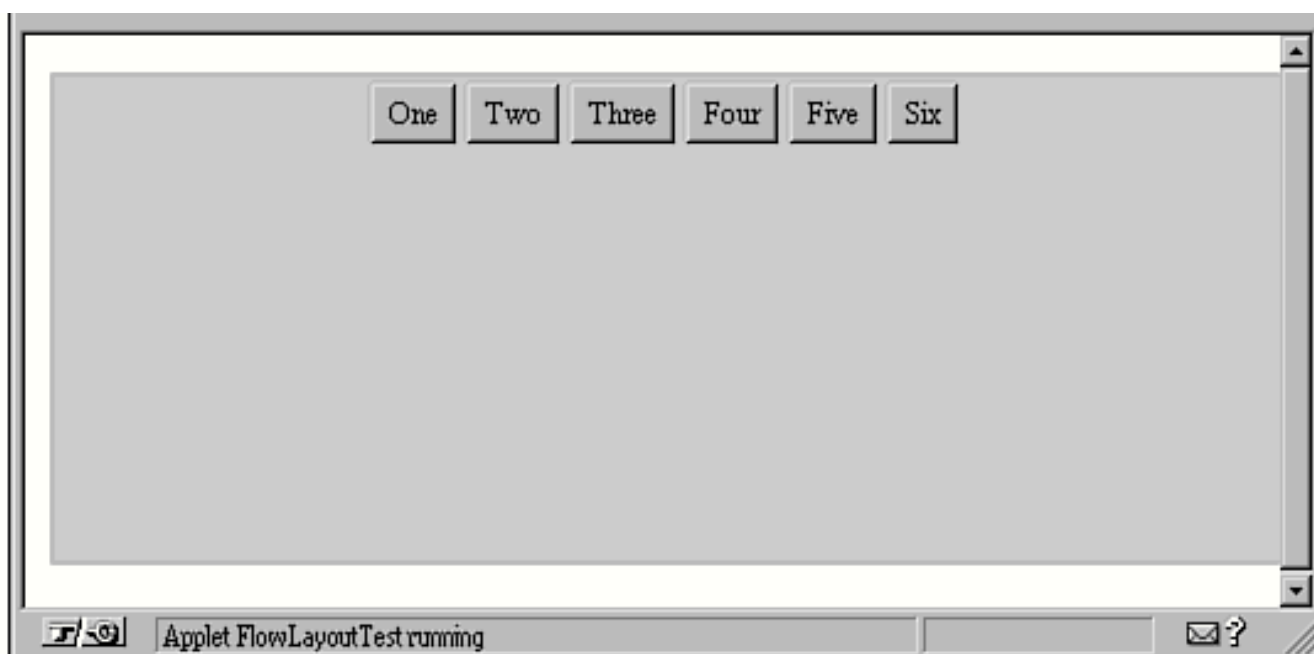
```
setLayout(new FlowLayout());
```

With the layout set, the order in which you add elements to the layout determines their position. The following code creates a simple row of six buttons in a centered flow layout.

```
import java.awt.*;

public class FlowLayoutTest extends java.applet.Applet {

    public void init() {
        setLayout(new FlowLayout());
        add(new Button("One"));
        add(new Button("Two"));
        add(new Button("Three"));
        add(new Button("Four"));
        add(new Button("Five"));
        add(new Button("Six"));
    }
}
```



To create a flow layout with an alignment other than centered, add the `FlowLayout.RIGHT` or `FlowLayout.LEFT` class variable as an argument:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

You can also set horizontal and vertical gap values by using flow layouts. The *gap* is the number of pixels between components in a panel; by default, the horizontal and vertical gap values are three pixels, which can be very close indeed. Horizontal gap spreads out components to the left and to the right; vertical gap spreads them to the top and bottom of each component. Add integer arguments to the flow layout constructor to increase the gap.

Figure shows the result of adding a gap of 30 points in the horizontal and 10 in the vertical directions, like this:

```
setLayout(new FlowLayout(FlowLayout.LEFT, 30, 10));
```



Grid Layouts

- *Grid layouts* offer more control over the placement of components inside a panel.
- Using a grid layout, you portion off the display area of the panel into rows and columns.
- Each component you then add to the panel is placed in a *cello* of the grid, starting from the top row and progressing through each row from left to right (here's where the order of calls to the `add()` method are very relevant to how the screen is laid out).
- To create a grid layout, indicate the number of rows and columns you want the grid to have when you create a new instance of the `GridLayout` class. Here's a grid layout with three rows and two columns.


```
import java.awt.*;

public class GridLayoutTest extends java.applet.Applet {

    public void init() {
        setLayout(new GridLayout(3,2);
        add(new Button("One"));
        add(new Button("Two"));
        add(new Button("Three"));
        add(new Button("Four"));
        add(new Button("Five"));
        add(new Button("Six"));
    }
}
```

Grid layouts can also have a horizontal and vertical gap between components. To create gaps, add those pixel values:

```
setLayout(new GridLayout(3, 3, 10, 30));
```

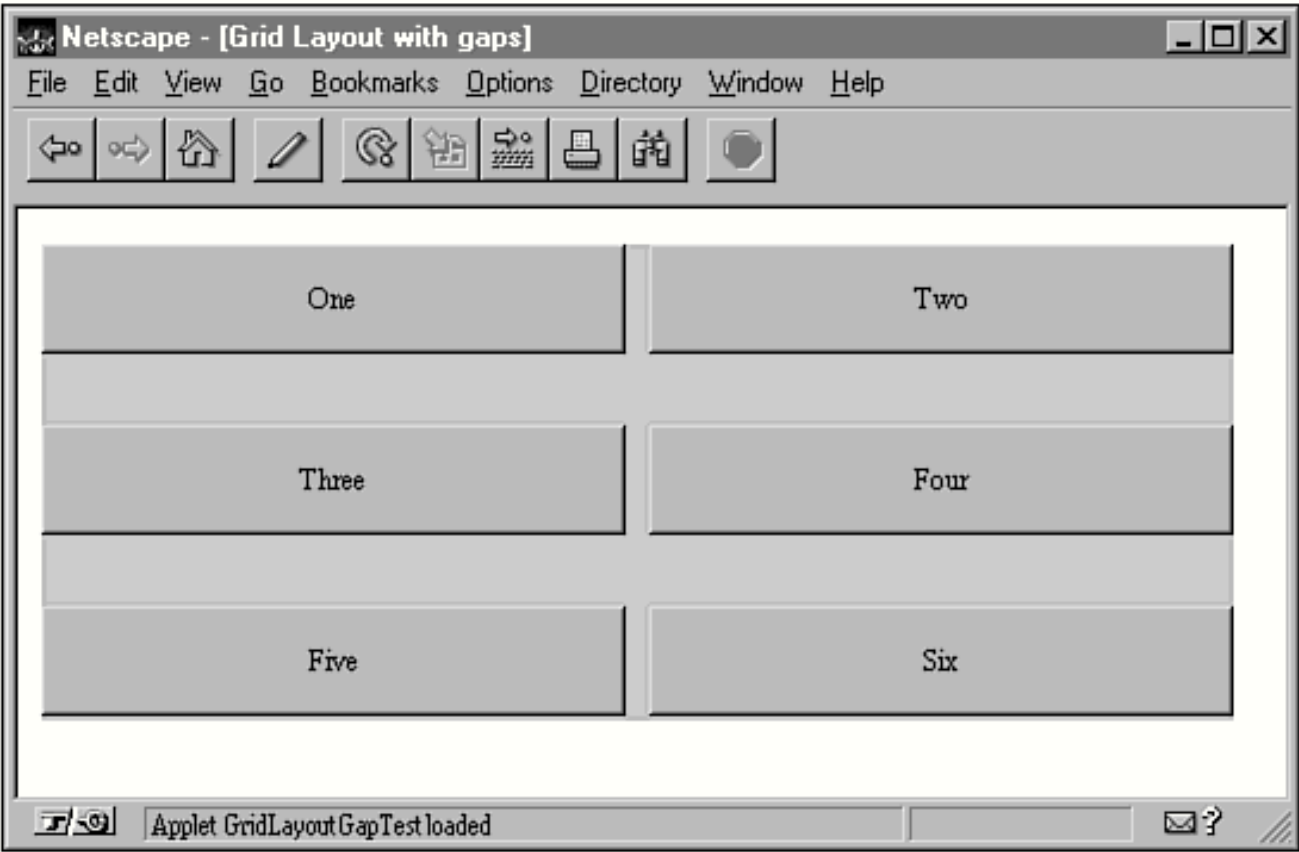


Figure shows a grid layout with a 10-pixel horizontal gap and a 30-pixel vertical gap.

Border Layouts

- *Border layouts* behave differently from flow and grid layouts.
- When you add a component to a panel that uses a border layout, you indicate its placement as a geographic direction: north, south, east, west, or center.

- The components around all the edges are laid out with as much size as they need; the component in the center, if any, gets any space left over.
- To use a border layout, you create it as you do the other layouts; then you add the individual components with a special `add()` method that has two arguments.
- The first argument is a string indicating the position of the component within the layout, and the second is the component to add:

```
add("North", new TextField("Title", 50));
```

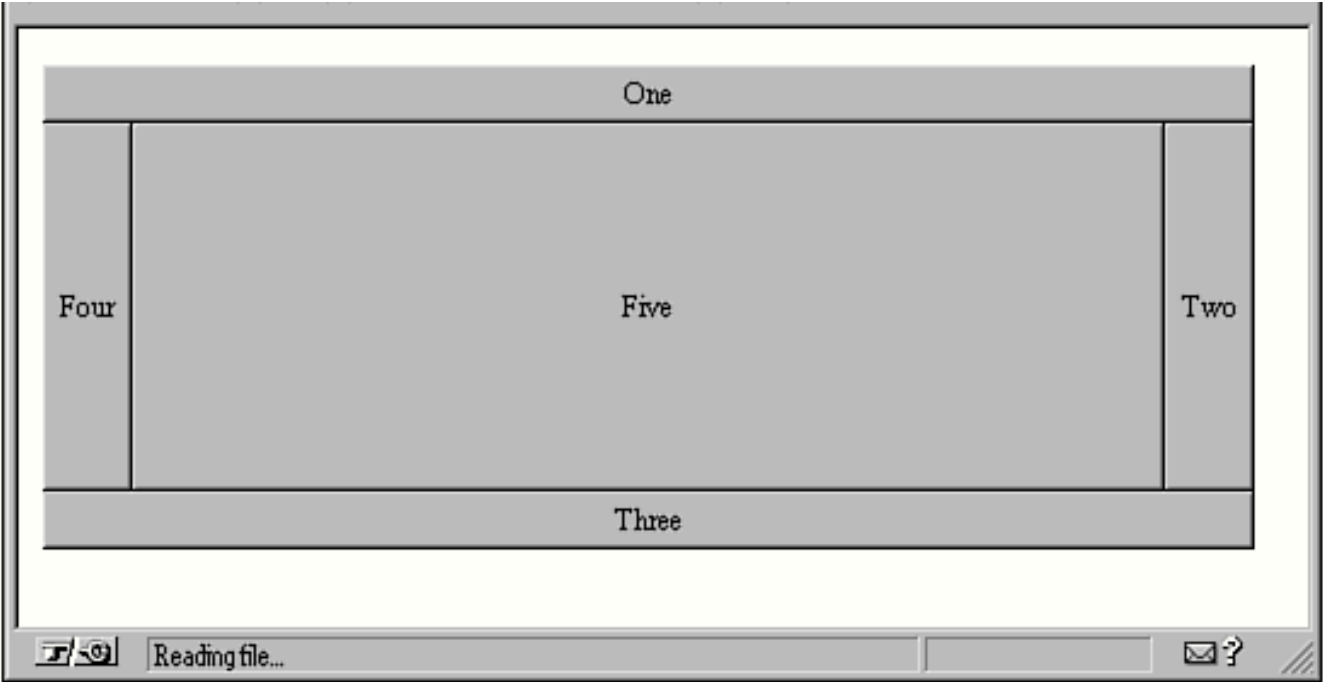
You can also use this form of `add()` for the other layout managers; the string argument will just be ignored if it's not needed.

Here's the code to generate the border layout shown in Figure:

```
import java.awt.*;

public class BorderLayoutTest extends java.applet.Applet {

    public void init() {
        setLayout(new BorderLayout());
        add("North", new Button("One"));
        add("East", new Button("Two"));
        add("South", new Button("Three"));
        add("West", new Button("Four"));
        add("Center", new Button("Five"));
        add(new Button("Six"));
    }
}
```



Border layouts can also have horizontal and vertical gaps. Note that the north and south components extend all the way to the edge of the panel, so the gap will result in less vertical space for the east, right, and center components. To add gaps to a border layout, include those pixel values in the constructor as with the other layout managers:

```
setLayout(new BorderLayout(10, 10));
```

Card Layouts

- *Card layouts* behave much differently from the other layouts.
- When you add components to one of the other layout managers, all those components appear on the screen at once. Card layouts are used to produce slide shows of components, one at a time. If you've ever used the HyperCard program on the Macintosh, or seen dialog boxes on windows with several different tabbed pages, you've worked with the same basic idea.
- When you create a card layout, the components you add to the outer panel will be other container components-usually other panels. You can then use different layouts for those individual cards so that each screen has its own look.
- *Cards*, in a card layout, are different panels added one at a time and displayed one at a time. If you think of a card file, you'll get the idea; only one card can be displayed at once, but you can switch between cards.
- When you add each card to the panel, you can give it a name. Then, to flip between the container cards, you can use methods defined in the `CardLayout` class to move to a named card, move forward or back, or move to the first card or to the last card. Typically you'll have a set of buttons that call these methods to make navigating the card layout easier.

Here's a simple snippet of code that creates a card layout containing three cards:

```
setLayout(new CardLayout());
//add the cards
Panel one = new Panel()
add("first", one);
Panel two = new Panel()
add("second", two);
Panel three = new Panel()
add("third", three);

// move around
show(this, "second"); //go to the card named "second"
show(this, "third");  //go to the card named "third"
previous(this);       //go back to the second card
first(this);          // got to the first card
```

Grid Bag Layouts

I've saved grid bag layouts for last because although they are the most powerful way of managing awt layout, they are also extremely complicated.

Using one of the other four layout managers, it can sometimes be difficult to get the exact layout you want without doing a lot of nesting of panels within panels. Grid bags provide a more general-purpose solution. Like grid layouts, *grid bag layouts* allow you to arrange your components in a grid-like layout. However, grid bag layouts also allow you to control the span of individual cells in the grid, the proportions between the rows and columns, and the arrangement of components inside cells in the grid.

To create a grid bag layout, you actually use two classes: `GridBagLayout`, which provides the overall layout manager, and `GridBagConstraints`, which defines the properties of each component in the grid-its placement, dimensions, alignment, and so on. It's the relationship between the grid bag, the constraints, and each component that defines the overall layout.

In its most general form, creating a grid bag layout involves the following steps:

- Creating a `GridBagLayout` object and defining it as the current layout manager, as you would any other layout manager
- Creating a new instance of `GridBagConstraints`
- Setting up the constraints for a component
- Telling the layout manager about the component and its constraints
- Adding the component to the panel

Here's some simple code that sets up the layout and then creates constraints for a single button (don't worry about the various values for the constraints; I'll cover these later on in this section):

```
// set up layout
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints constraints = new GridBagConstraints();
setLayout(gridbag);
```

```
// define constraints for the button
Button b = new Button("Save");
constraints.gridx = 0;
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 1;
constraints.weightx = 30;
constraints.weighty = 30;
constraints.fill = GridBagConstraints.NONE;
```

```
constraints.anchor = GridBagConstraints.CENTER;
```

```
// attach constraints to layout, add button  
gridbag.setConstraints(b, constraints);  
add(b);
```

Retrieving and Using Images

- Basic image handling in Java is easy. The `Image` class in the `java.awt` package provides abstract methods to represent common image behavior, and special methods defined in `Applet` and `Graphics` give you everything you need to load and display images in your applet as easily as drawing a rectangle.

Getting Images

- To display an image in your applet, you first must load that image over the Net into your Java program. Images are stored as separate files from your Java class files, so you have to tell Java where to find them.
- The `Applet` class provides a method called `getImage()`, which loads an image and automatically creates an instance of the `Image` class for you. To use it, all you have to do is import the `java.awt.Image` class into your Java program, and then give `getImage` the URL of the image you want to load. There are two ways of doing the latter step:

The `getImage()` method with a single argument (an object of type `URL`) retrieves the image at that URL.

The `getImage()` method with two arguments: the base URL (also a `URLObject`) and a string representing the path or filename of the actual image (relative to the base).

The latter form, therefore, is usually the one to use. The `Applet` class also provides two methods that will help with the base URL argument to `getImage()`:

The `getDocumentBase()` method returns a `URLObject` representing the directory of the HTML file that contains this applet. So, for example, if the HTML file is located at `http://www.myserver.com/htmlfiles/javahtml/`, `getDocumentBase()` returns a `URL` pointing to that path.

The `getCodeBase()` method returns a string representing the directory in which this applet is contained—which may or may not be the same directory as the HTML file, depending on whether the `CODEBASE` attribute in `<APPLET>` is set or not.

- Whether you use `getDocumentBase()` or `getCodeBase()` depends on whether your images are relative to your HTML files or relative to your Java class files. Use whichever one applies better to your situation.
- Note that either of these methods is more flexible than hard-coding a URL or pathname into the `getImage()` method; using either `getDocumentBase()` or `getCodeBase()` enables you to move your HTML files and applets around and Java can still find your images
- Here are a few examples of `getImage`, to give you an idea of how to use it. This first call to `getImage()` retrieves the file at that specific URL (<http://www.server.com/files/image.gif>). If any part of that URL changes, you have to recompile your Java applet to take into account the new path:

```
Image img = getImage(
    new URL("    http://www.server.com/files/image.gif    "));
```

In the following form of `getImage`, the `image.gif` file is in the same directory as the HTML files that refer to this applet:

```
Image img = getImage(getDocumentBase(), "image.gif")
```

In this similar form, the file `image.gif` is in the same directory as the applet itself:

```
Image img = getImage(getCodeBase(), "image.gif")
```

If you have lots of image files, it's common to put them into their own subdirectory. This form of `getImage()` looks for the file `image.gif` in the directory `images`, which, in turn, is in the same directory as the Java applet:

```
Image img = getImage(getCodeBase(), "images/image.gif")
```

If `getImage()` can't find the file indicated, it returns `null`. `drawImage()` on a `null` image will simply draw nothing. Using a `null` image in other ways will probably cause an error.

Note

Currently, Java supports images in the GIF and JPEG formats. Other image formats may be available later; however, for now, your images should be in either GIF or JPEG.

Drawing Images

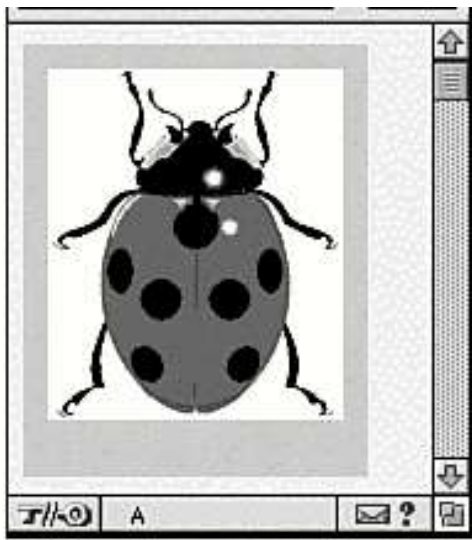
- All that stuff with `getImage()` does nothing except go off and retrieve an image and stuff it into an instance of the `Image` class. Now that you have an image, you have to do something with it.
- The most likely thing you're going to want to do with an image is display it as you would a rectangle or a text string. The `Graphics` class provides two methods to do just this, both called `drawImage()`.
- The first version of `drawImage()` takes four arguments: the image to display, the `x` and `y` positions of the top left corner, and `this`:

```
public void paint() {  
    g.drawImage(img, 10, 10, this);  
}
```

- This first form does what you would expect it to: It draws the image in its original dimensions with the top-left corner at the given `x` and `y` positions.
- Listing below shows the code for a very simple applet that loads an image called `ladybug.gif` and displays it. Figure 11.1 shows the obvious result.

Listing . The Ladybug applet.

```
1:import java.awt.Graphics;  
2:import java.awt.Image;  
3:  
4:public class LadyBug extends java.applet.Applet {  
5:  
6:    Image bugimg;  
7:  
8:    public void init() {  
9:        bugimg = getImage(getCodeBase(),  
10:        "images/ladybug.gif");  
11:    }  
12:  
13:    public void paint(Graphics g) {  
14:        g.drawImage(bugimg, 10, 10,this);  
15:    }  
16:}
```

- In this example the instance variable `bugimg` holds the ladybug image, which is loaded in the `init()` method. The `paint()` method then draws that image on the screen.
- The second form of `drawImage()` takes six arguments: the image to draw, the `x` and `y` coordinates of the top-left corner, a width and height of the image bounding box, and `this`.
- If the width and height arguments for the bounding box are smaller or larger than the actual image, the image is automatically scaled to fit.
- By using those extra arguments, you can squeeze and expand images into whatever space you need them to fit in (keep in mind, however, that there may be some image degradation from scaling it smaller or larger than its intended size).
- One helpful hint for scaling images is to find out the size of the actual image that you've loaded, so you can then scale it to a specific percentage and avoid distortion in either direction.
- Two methods defined for the `Image` class can give you that information:
`getWidth()` and `getHeight()`.
- Both take a single argument, an instance of `ImageObserver`, which is used to track the loading of the image (more about this later). Most of the time, you can use just `this` as an argument to either `getWidth()` or `getHeight()`.

If you stored the ladybug image in a variable called `bugimg`, for example, this line returns the width of that image, in pixels:

```
theWidth = bugimg.getWidth(this);
```

Technical Note

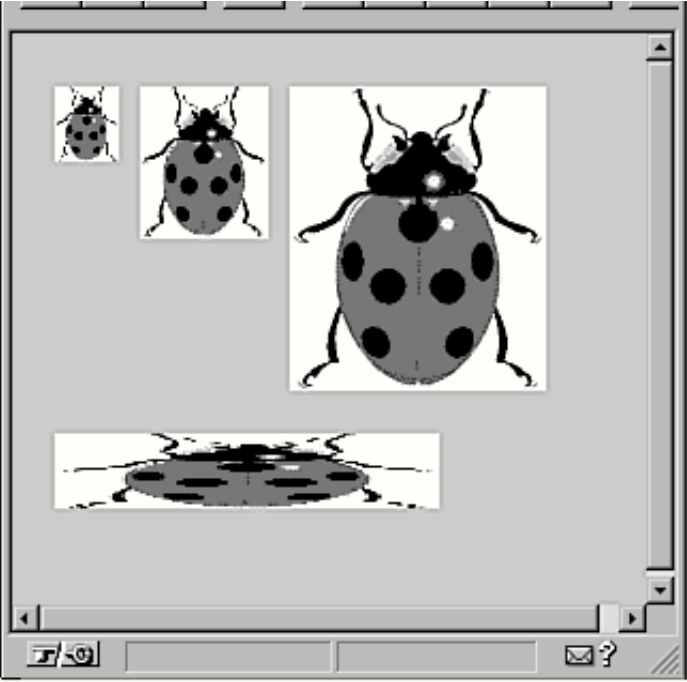
Here's another case where, if the image isn't loaded all the way, you may get different results. Calling `getWidth()` or `getHeight()` before the image has fully loaded will result in values of `-1` for each one. Tracking image loading with image observers can help you keep track of when this information appears.

Listing shows another use of the ladybug image, this time scaled several times to different sizes .

Listing: More ladybugs, scaled.

```
1: import java.awt.Graphics;
2: import java.awt.Image;
3:
4: public class LadyBug2 extends java.applet.Applet {
5:
6:     Image bugimg;
7:
8:     public void init() {
9:         bugimg = getImage(getCodeBase(),
10:             "images/ladybug.gif");
11:     }
12:
13:     public void paint(Graphics g) {
14:         int iwidth = bugimg.getWidth(this);
15:         int iheight = bugimg.getHeight(this);
16:         int xpos = 10;
17:
18:         // 25 %
19:         g.drawImage(bugimg, xpos, 10,
20:             iwidth / 4, iheight / 4, this);
21:
22:         // 50 %
23:         xpos += (iwidth / 4) + 10;
24:         g.drawImage(bugimg, xpos , 10,
25:             iwidth / 2, iheight / 2, this);
26:
27:         // 100%
```

```
28:         xpos += (iwidth / 2) + 10;
29:         g.drawImage(bugimg, xpos, 10, this);
30:
31:         // 150% x, 25% y
32:         g.drawImage(bugimg, 10, iheight + 30,
33:             (int)(iwidth * 1.5), iheight / 4, this);
34:     }
35: }
```



Event Handling in Java

- Java events are part of the Java awt (Abstract Windowing Toolkit) package.
- An event is the way that the awt communicates to you, as the programmer, and to other Java awt components that *something* has happened. That something can be input from the user (mouse movements or clicks, keypresses), changes in the system environment (a window opening or closing, the window being scrolled up or down), or a host of other things that might, in some way, affect the operation of the program.
- Some events are handled by the awt or by the environment your applet is running in (the browser) without you needing to do anything. `paint()` methods, for example, are generated and handled by the environment-all you have to do is tell the awt what you want painted when it gets to your part of the window.
- However, you may need to know about some events, such as a mouse click inside the boundaries of your applet. By writing your Java programs to handle these kinds of events, you can get input from the user and have your applet change its behavior based on that input.

This section discusses about managing simple events, including the following basics:

- Mouse clicks
- Mouse movements, including mouse dragging
- Keyboard actions

`handleEvent()` **method**, which is the basis for collecting, handling, and passing on events of all kinds from your applet to other components of the window or of your applet itself.

Mouse Clicks

- Mouse-click events occur when your user clicks the mouse somewhere in the body of your applet. You can intercept mouse clicks to do very simple things-for example, to toggle the sound on and off in your applet, to move to the next slide in a presentation, or to clear the screen and start over-or you can use mouse clicks in conjunction with mouse movements to perform more complex motions inside your applet.

Mouse Down and Mouse Up Events

- When you click the mouse once, the awt generates two events: a mouse down event when the mouse button is pressed and a mouse up event when the button is released.
- Why two individual events for a single mouse action? Because you may want to do different things for the "down" and the "up." For example, look at a pull-down menu.
- The mouse down extends the menu, and the mouse up selects an item (with mouse drags between-but you'll learn about that one later). If you have only one event for both actions (mouse up and mouse down), you cannot implement that sort of user interaction.
- Handling mouse events in your applet is easy-all you have to do is override the right method definition in your applet. That method will be called when that particular event occurs. Here's an example of the method signature for a mouse down event:

```
public boolean mouseDown(Event evt, int x, int y) {  
...  
}
```

The `mouseDown()` method (and the `mouseUp()` method as well) takes three parameters: the event itself and the x and y coordinates where the mouse down or mouse up event occurred.

- The `evt` argument is an instance of the class `Event`. All system events generate an instance of the `Event` class, which contains information about where and when the event took place, the kind of event it is, and other information that you might want to know about this event
- The x and the y coordinates of the event, as passed in through the `x` and `y` arguments to the `mouseDown()` method, are particularly nice to know because you can use them to determine precisely where the mouse click took place.
- So, for example, if the mouse down event were over a graphical button, you could activate that button.

For example, here's a simple method that prints out information about a mouse down when it occurs:

```
public boolean mouseDown(Event evt, int x, int y)  
{  
System.out.println("Mouse down at " + x + ", " + y);  
return true;  
}
```

By including this method in your applet, every time your user clicks the mouse inside your applet, this message will get printed. The awt system calls each of these methods when the actual event takes place.

- Note that this method, `mouseUp()`, returns a boolean value instead of not returning anything (`void`). Having an event handler method return `true` or `false` determines whether a given component can intercept an event or whether it needs to pass it on to the enclosing component.
- The general rule is that if your method intercepts and does something with the event, it should return `true` .
- If for any reason the method doesn't do anything with that event, it should return `false` so that other components in the system can have a chance to see that event.
- In most of the examples in today's lesson, you'll be intercepting simple events, so most of the methods here will return `true` . Tomorrow you'll learn about nesting components and passing events up the component hierarchy.
- The second half of the mouse click is the `mouseUp()` method, which is called when the mouse button is released. To handle a mouse up event, add the `mouseUp()` method to your applet: `mouseUp()` looks just like `mouseDown()` :

```
public boolean mouseUp(Event evt, int x, int y) {
    ....
}
```

An Example: Spots

In this section you'll create an example of an applet that uses mouse events-mouse down events in particular. The Spots applet starts with a blank screen and then sits and waits. When you click the mouse on that screen, a blue dot is drawn. You can place up to 10 dots on the screen. Figure below shows the Spots applet.

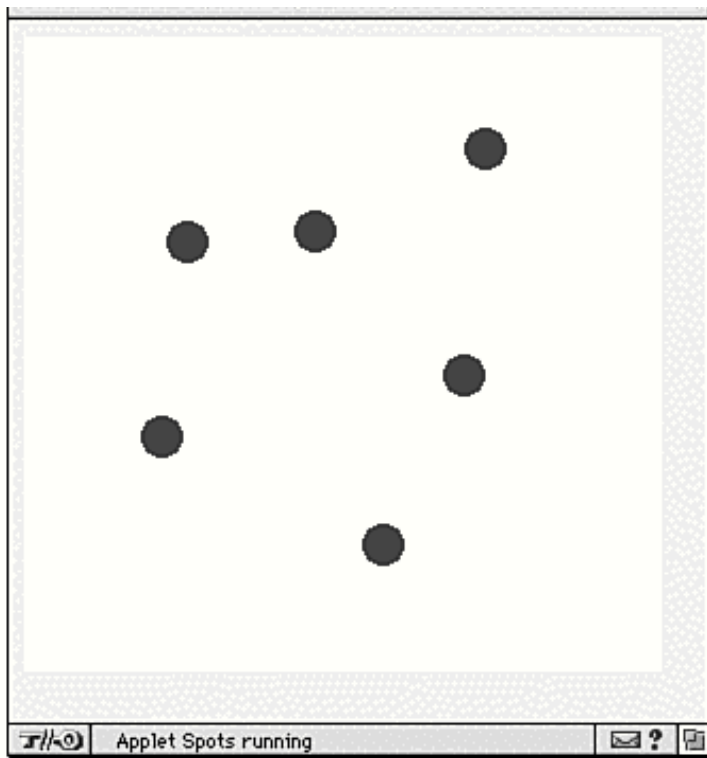
Let's start from the beginning and build this applet, starting from the initial class definition:

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;

public class Spots extends java.applet.Applet {

    final int MAXSPOTS = 10;
    int xspots[] = new int[MAXSPOTS];
    int yspots[] = new int[MAXSPOTS];
    int currspots = 0;

}
```



- This class uses three other awt classes: `Graphics`, `Color`, and `Event`. That last class, `Event`, needs to be imported in any applets that use events. The class has four instance variables: a constant to determine the maximum number of spots that can be drawn, two arrays to store the x and y coordinates of the spots that have already been drawn, and an integer to keep track of the number of the current spot.
- Let's start by adding the `init()` method, which does only one thing: set the background color to `white`:

```
public void init() {
setBackground(Color.white);
}
```

- We've set the background here in `init()` instead of in `paint()` as you have in past examples because you need to set the background only once. Because `paint()` is called repeatedly each time a new spot is added, setting the background in the `paint()` method unnecessarily slows down that method. Putting it here is a much better idea.

The main action of this applet occurs with the `mouseDown()` method, so let's add that one now:

```
public boolean mouseDown(Event evt, int x, int y) {
if (currspots < MAXSPOTS) {
addspot(x,y);
return true;
}
else {
System.out.println("Too many spots.");
return false;
}
}
```

- When the mouse click occurs, the `mouseDown()` method tests to see whether there are fewer than 10 spots. If so, it calls the `addspot()` method (which you'll write soon) and returns `true` (the mouse down event was intercepted and handled). If not, it just prints an error message and returns `false`.
- What does `addspot()` do? It adds the coordinates of the spot to the arrays that store the coordinates, increments the `currspots` variable, and then calls `repaint()`:

```
void addspot(int x, int y) {
    xspots[currspots] = x;
    yspots[currspots] = y;
    currspots++;
    repaint();
}
```

- You may be wondering why you have to keep track of all the past spots in addition to the current spot. It's because of `repaint()`: Each time you paint the screen, you have to paint all the old spots in addition to the newest spot. Otherwise, each time you painted a new spot, the older spots would get erased. Now, on to the `paint()` method:

```
public void paint(Graphics g) {
    g.setColor(Color.blue);
    for (int i = 0; i < currspots; i++) {
        g.fillOval(xspots[i] - 10, yspots[i] - 10, 20, 20);
    }
}
```

- Inside `paint()`, you just loop through the spots you've stored in the `xspots` and `yspots` arrays, painting each one (actually, painting them a little to the right and upward so that the spot is painted around the mouse pointer rather than below and to the right).
- That's it! That's all you need to create an applet that handles mouse clicks. Everything else is handled for you. You have to add the appropriate behavior to `mouseDown()` or `mouseUp()` to intercept and handle that event. Listing shows the full text for the Spots applet.

Listing . The Spots applet.

```
1: import java.awt.Graphics;
2: import java.awt.Color;
3: import java.awt.Event;
4:
5: public class Spots extends java.applet.Applet {
6:
7:     final int MAXSPOTS = 10;
8:     int xspots[] = new int[MAXSPOTS];
9:     int yspots[] = new int[MAXSPOTS];
10:     int currspots = 0;
```



```

11:
12:     public void init() {
13:         setBackground(Color.white);
14:     }
15:
16:     public boolean mouseDown(Event evt, int x, int y) {
17:         if (currspots < MAXSPOTS) {
18:             addspot(x,y);
19:             return true;
20:         }
21:         else {
22:             System.out.println("Too many spots.");
23:             return false;
24:         }
25:     }
26:
27:     void addspot(int x,int y) {
28:         xspots[currspots] = x;
29:         yspots[currspots] = y;
30:         currspots++;
31:         repaint();
32:     }
33:
34:     public void paint(Graphics g) {
35:         g.setColor(Color.blue);
36:         for (int i = 0; i < currspots; i++) {
37:             g.fillOval(xspots[i] - 10, yspots[i] - 10, 20, 20);
38:         }
39:     }
40: }

```

Double-Clicks

- What if the mouse event you're interested in is more than a single mouse click-what if you want to track double- or triple-clicks? The Java `Event` class provides a variable for tracking this information, called `clickCount`. `clickCount` is an integer representing the number of consecutive mouse clicks that have occurred (where "consecutive" is usually determined by the operating system or the mouse hardware). If you're interested in multiple mouse clicks in your applets, you can test this value in the body of your `mouseDown()` method, like this:

```

public boolean mouseDown(Event evt, int x, int y) {
switch (evt.clickCount) {
case 1: // single-click
case 2: // double-click
case 3: // triple-click
....
}
}

```

Mouse Movements

Every time the mouse is moved a single pixel in any direction, a mouse move event is generated. There are two mouse movement events: mouse drags, where the movement occurs with the mouse button pressed down, and plain mouse movements, where the mouse button isn't pressed.

To manage mouse movement events, use the `mouseDrag()` and `mouseMove()` methods.

Mouse Drag and Mouse Move Events

The `mouseDrag()` and `mouseMove()` methods, when included in your applet code, intercept and handle mouse movement events. Mouse move and mouse drag events are generated for every pixel change the mouse moves, so a mouse movement from one side of the applet to the other may generate hundreds of events. The `mouseMove()` method, for plain mouse pointer movements without the mouse button pressed, looks much like the mouse-click methods:

```
public boolean mouseMove(Event evt, int x, int y) {  
    ...  
}
```

The `mouseDrag()` method handles mouse movements made with the mouse button pressed down (a complete dragging movement consists of a mouse down event, a series of mouse drag events for each pixel the mouse is moved, and a mouse up when the button is released). The `mouseDrag()` method looks like this:

```
public boolean mouseDrag(Event evt, int x, int y) {  
    ...  
}
```

Note that for both the `mouseMove()` and `mouseDrag()` methods, the arguments for the x and y coordinates are the new location of the mouse, not its starting location.

Mouse Enter and Mouse Exit Events

Finally, there are the `mouseEnter()` and `mouseExit()` methods. These two methods are called when the mouse pointer enters or exits an applet or a portion of that applet. (In case you're wondering why you might need to know this, it's more useful on awt components that you might put inside an applet.)

Both `mouseEnter()` and `mouseExit()` have signatures similar to the mouse click methods—three arguments: the event object and the x and y coordinates of the point where the mouse entered or exited the applet. These examples show the signatures for `mouseEnter()` and `mouseExit()`:

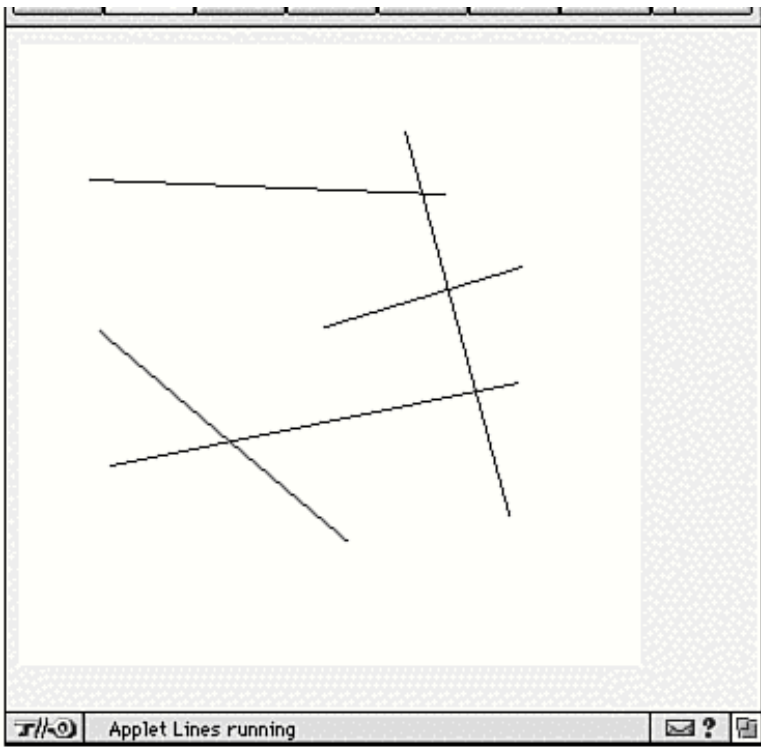
```
public boolean mouseEnter(Event evt, int x, int y) {  
    ...  
}  
  
public boolean mouseExit(Event evt, int x, int y) {  
    ...  
}
```

An Example: Drawing Lines

Examples always help to make concepts more concrete. In this section you'll create an applet that enables you to draw straight lines on the screen by dragging from the startpoint to the endpoint. Figure 12.2 shows the applet at work.

As with the Spots applet (on which this applet is based), let's start with the basic definition and work our way through it, adding the appropriate methods to build the applet. Here's a simple class definition for the Lines applet, with a number of initial instance variables and a simple `init()` method:

```
import java.awt.Graphics;  
import java.awt.Color;  
import java.awt.Event;  
import java.awt.Point;  
  
public class Lines extends java.applet.Applet {  
  
    final int MAXLINES = 10;  
    Point starts[] = new Point[MAXLINES]; // starting points  
    Point ends[] = new Point[MAXLINES];   // ending points  
    Point anchor;    // start of current line  
    Point currentpoint; // current end of line  
    int currline = 0; // number of lines  
  
    public void init() {  
        setBackground(Color.white);  
    }  
}
```



- This applet adds a few more things than Spots. Unlike Spots, which keeps track of individual integer coordinates, this one keeps track of `Point` objects. Points represent an x and a y coordinate, encapsulated in a single object. To deal with points, you import the `Point` class and set up a bunch of instance variables that hold points:

`starts` array holds points representing the starts of lines already drawn.

`ends` array holds the endpoints of those same lines.

`anchor` holds the starting point of the line currently being drawn.

`currentpoint` holds the current endpoint of the line currently being drawn.

`currline` holds the current number of lines (to make sure you don't go over `MAXLINES` and to keep track of which line in the array to access next).

- Finally, the `init()` method, as in the Spots applet, sets the background of the applet to `white`.
- The three main events this applet deals with are `mouseDown()`, to set the anchor point for the current line, `mouseDrag()`, to animate the current line as it's being drawn, and `mouseUp()`, to set the ending point for the new line. Given that you have instance variables to hold each of these values, it's merely a matter of plugging the right variables into the right methods. Here's `mouseDown()`, which sets the anchor point (but only if we haven't exceeded the maximum number of lines):

```
public boolean mouseDown(Event evt, int x, int y) {
    if (currline < MAXLINES) {
        anchor = new Point(x,y);
        return true;
    }
    else {
        System.out.println("Too many lines.");
        return false;
    }
}
```

```
}  
}
```

- While the mouse is being dragged to draw the line, the applet animates the line being drawn. As you drag the mouse around, the new line moves with it from the anchor point to the tip of the mouse.
- The `mouseDrag()` event contains the current point each time the mouse moves, so use that method to keep track of the current point (and to repaint for each movement so the line "animates"). Note that if we've exceeded the maximum number of lines, we won't want to do any of this. Here's the `mouseDrag()` method to do all those things:

```
public boolean mouseDrag(Event evt, int x, int y) {  
    if (currline < MAXLINES) {  
        currentpoint = new Point(x,y);  
        repaint();  
        return true;  
    }  
    else return false;  
}
```

The new line doesn't get added to the arrays of old lines until the mouse button is released. Here's

`mouseUp()` , which tests to make sure you haven't exceeded the maximum number of lines before calling the `addline()` method (described next):

```
public boolean mouseUp(Event evt, int x, int y) {  
    if (currline < MAXLINES) {  
        addline(x,y);  
        return true;  
    }  
    else return false;  
}
```

The `addline()` method is where the arrays of starting and ending points get updated and where the applet is repainted to take the new line into effect:

```
void addline(int x,int y) {  
    starts[currline] = anchor;  
    ends[currline] = new Point(x,y);  
    currline++;  
    currentpoint = null;  
    anchor = null;  
    repaint();  
}
```

- Note that in this method you also set `currentpoint` and `anchor` to `null` . Why? Because the current line you were drawing is over. By setting these variables to `null` , you can test for that value in the `paint()` method to see whether you need to draw a current line.

Painting the applet means drawing all the old lines stored in the `starts` and `ends` arrays, as well as drawing the current line in progress (whose endpoints are in `anchor` and `currentpoint`, respectively). To show the animation of the current line, draw it in blue. Here's the `paint()` method for the Lines applet:

```
public void paint(Graphics g) {

    // Draw existing lines
    for (int i = 0; i < currline; i++) {
        g.drawLine(starts[i].x, starts[i].y,
            ends[i].x, ends[i].y);
    }

    // Draw current line
    g.setColor(Color.blue);
    if (currentpoint != null)
        g.drawLine(anchor.x, anchor.y,
            currentpoint.x, currentpoint.y);
}
```

In `paint()`, when you're drawing the current line, you test first to see whether `currentpoint` is `null`. If it is, the applet isn't in the middle of drawing a line, so there's no reason to try drawing a line that doesn't exist. By testing for `currentpoint` (and by setting `currentpoint` to `null` in the `addline()` method), you can paint only what you need.

That's it—just 60 lines of code and a few basic methods, and you have a very basic drawing application in your Web browser. Listing below shows the full text of the Lines applet so that you can put the pieces together.

Listing The Lines applet.

```
1: import java.awt.Graphics;
2: import java.awt.Color;
3: import java.awt.Event;
4: import java.awt.Point;
5:
6: public class Lines extends java.applet.Applet {
7:
8:     final int MAXLINES = 10;
9:     Point starts[] = new Point[MAXLINES]; // starting points
10:    Point ends[] = new Point[MAXLINES];    // endingpoints
11:    Point anchor;    // start of current line
12:    Point currentpoint; // current end of line
13:    int currline = 0; // number of lines
14:
15:    public void init() {
16:        setBackground(Color.white);
17:    }
18:
19:    public boolean mouseDown(Event evt, int x, int y) {
20:        if (currline < MAXLINES) {
21:            anchor = new Point(x,y);
```

```

22:         return true;
23:     }
24:     else {
25:         System.out.println("Too many lines.");
26:         return false;
27:     }
28: }
29:
30: public boolean mouseUp(Event evt, int x, int y) {
31:     if (currline < MAXLINES) {
32:         addline(x,y);
33:         return true;
34:     }
35:     else return false;
36: }
37:
38: public boolean mouseDrag(Event evt, int x, int y) {
39:     if (currline < MAXLINES) {
40:         currentpoint = new Point(x,y);
41:         repaint();
42:         return true;
43:     }
44:     else return false;
45: }
46:
47: void addline(int x,int y) {
48:     starts[currline] = anchor;
49:     ends[currline] = new Point(x,y);
50:     currline++;
51:     currentpoint = null;
52:     anchor = null;
53:     repaint();
54: }
55:
56: public void paint(Graphics g) {
57:
58:     // Draw existing lines
59:     for (int i = 0; i < currline; i++) {
60:         g.drawLine(starts[i].x, starts[i].y,
61:             ends[i].x, ends[i].y);
62:     }
63:
64:     // draw current line
65:     g.setColor(Color.blue);
66:     if (currentpoint != null)
67:         g.drawLine(anchor.x,anchor.y,
68:             currentpoint.x,currentpoint.y);
69: }
70: }

```

Keyboard Events

- A keyboard event is generated whenever a user presses a key on the keyboard. By using keyboard events, you can get hold of the values of the keys the user pressed to perform an action or merely to get character input from the users of your applet.

The `keyDown()` and `keyUp()` Methods

To capture a keyboard event, use the `keyDown()` method:

```
public boolean keyDown(Event evt, int key) {  
...  
}
```

- The keys generated by key down events (and passed into `keyDown()` as the `key` argument) are integers representing Unicode character values, which include alphanumeric characters, function keys, tabs, returns, and so on. To use them as characters (for example, to print them), you need to cast them to characters:

```
currentchar = (char)key;
```

- Here's a simple example of a `keyDown()` method that does nothing but print the key you just typed in both its Unicode and character representation (it can be fun to see which key characters produce which values):

```
public boolean keyDown(Event evt, int key) {  
    System.out.println("ASCII value: " + key);  
    System.out.println("Character: " + (char)key);  
    return true;  
}
```

As with mouse clicks, each key down event also has a corresponding key up event. To intercept key up events, use the `keyUp()` method:

```
public boolean keyUp(Event evt, int key) {  
...  
}
```

Default Keys

- The `Event` class provides a set of class variables that refer to several standard nonalphanumeric keys, such as the arrow and function keys. If your applet's interface uses these keys, you can provide more readable code by testing for these names in your `keyDown()` method rather than testing for their numeric values (and you're also more likely to be cross-platform if you use these variables).
- For example, to test whether the up arrow was pressed, you might use the following snippet of code:

```
if (key == Event.UP) {  
...  
}
```
- Because the values these class variables hold are integers, you also can use the `switch` statement to test for them.
- Table 12.1 shows the standard event class variables for various keys and the actual keys they represent.

Table : Standard keys defined by the Event class.
Class Variable Represented Key

Event.HOME	The Home key
Event.END	The End key
Event.PGUP	The Page Up key
Event.PGDN	The Page Down key
Event.UP	The up arrow
Event.DOWN	The down arrow
Event.LEFT	The left arrow
Event.RIGHT	The right arrow
Event.f1	The f1 key
Event.f2	The f2 key
Event.f3	The f3 key
Event.f4	The f4 key
Event.f5	The f5 key
Event.f6	The f6 key
Event.f7	The f7 key
Event.f8	The f8 key
Event.f9	The f9 key
Event.f10	The f10 key
Event.f11	The f11 key
Event.f12	The f12 key

An Example: Entering, Displaying, and Moving Characters

- Let's look at an applet that demonstrates keyboard events. With this applet, you type a character, and that character is displayed in the center of the applet window. You then can move that character around on the screen with the arrow keys. Typing another character at any time changes the character as it's currently displayed. Figure below shows an example.

Note

To get this applet to work, you might have to click once with the mouse on it in order for the keys to show up. This is to make sure the applet has the keyboard focus (that is, that its actually listening when you type characters on the keyboard).

- This applet is actually less complicated than the previous applets you've used. This one has only three methods: `init()`, `keyDown()`, and `paint()`. The instance variables are also simpler because the only things you need to keep track of are the x and y positions of the current character and the values of that character itself. Here's the initial class definition:

```
import java.awt.Graphics;
import java.awt.Event;
import java.awt.Font;
import java.awt.Color;

public class Keys extends java.applet.Applet {

    char currkey;
    int currx;
    int curry;
}
```

- Let's start by adding an `init()` method. Here, `init()` is responsible for three things: setting the background color, setting the applet's font (here, 36-point Helvetica bold), and setting the beginning position for the character (the middle of the screen, minus a few points to nudge it up and to the right):

```
public void init() {
    currx = (size().width / 2) - 8;
    curry = (size().height / 2) - 16;
    setBackground(Color.white);
    setFont(new Font("Helvetica", Font.BOLD, 36));
}
```

- Because this applet's behavior is based on keyboard input, the `keyDown()` method is where most of the work of the applet takes place:

```
public boolean keyDown(Event evt, int key) {
    switch (key) {
        case Event.DOWN:
            curry += 5;
            break;
        case Event.UP:
            curry -= 5;
            break;
        case Event.LEFT:
            currx -= 5;
            break;
        case Event.RIGHT:
            currx += 5;
            break;
        default:
            currkey = (char)key;
    }
    repaint();
    return true;
}
```

- In the center of the `keyDown()` applet is a `switch` statement that tests for different key events. If the event is an arrow key, the appropriate change is made to the character's position. If the event is any other key, the character itself is changed (that's the default part of the `switch`). The method finishes up with a `repaint()` and returns `true`.
- The `paint()` method here is almost trivial; just display the current character at the current position. However, note that when the applet starts up, there's no initial character and nothing to draw, so you have to take that into account. The `currkey` variable is initialized to `0`, so you paint the applet only if `currkey` has an actual value:

```
public void paint(Graphics g) {
    if (currkey != 0) {
        g.drawString(String.valueOf(currkey), currx, curry);
    }
}
```

Listing shows the complete source code for the Keys applet.

Listing : The Keys applet.

```
1: import java.awt.Graphics;
2: import java.awt.Event;
3: import java.awt.Font;
4: import java.awt.Color;
5:
```

```

6: public class Keys extends java.applet.Applet {
7:
8:     char currkey;
9:     int currx;
10:    int curry;
11:
12:    public void init() {
13:        currx = (size().width / 2) -8;  // default
14:        curry = (size().height / 2) -16;
15:
16:        setBackground(Color.white);
17:        setFont(new Font("Helvetica",Font.BOLD,36));
18:    }
19:
20:    public boolean keyDown(Event evt, int key) {
21:        switch (key) {
22:            case Event.DOWN:
23:                curry += 5;
24:                break;
25:            case Event.UP:
26:                curry -= 5;
27:                break;
28:            case Event.LEFT:
29:                currx -= 5;
30:                break;
31:            case Event.RIGHT:
32:                currx += 5;
33:                break;
34:            default:
35:                currkey = (char)key;
36:        }
37:
38:        repaint();
39:        return true;
40:    }
41:
42:    public void paint(Graphics g) {
43:        if (currkey != 0) {
44:            g.drawString(String.valueOf(currkey), currx,curry);
45:        }
46:    }
47: }

```

Testing for Modifier Keys and Multiple Mouse Buttons

- Shift, Control (Ctrl), and Meta are modifier keys. They don't generate key events themselves, but when you get an ordinary mouse or keyboard event, you can test to see whether those modifier keys were held down when the event occurred.
- Sometimes it may be obvious-shifted alphanumeric keys produce different key events than unshifted ones, for example. For other events, however-mouse events in particular-you may want to handle an event with a modifier key held down differently from a regular version of that event.

Note

The Meta key is commonly used on UNIX systems; it's usually mapped to Alt on pc keyboards and Command (apple) on Macintoshes.

- The `Event` class provides three methods for testing whether a modifier key is held down: `shiftDown()`, `metaDown()`, and `controlDown()`. All return boolean values based on whether that modifier key is indeed held down.
- You can use these three methods in any of the event- handling methods (mouse or keyboard) by calling them on the event object passed into that method:

```
public boolean mouseDown(Event evt, int x, int y ) {  
    if (evt.shiftDown())  
        // handle shift-click  
    else // handle regular click  
}
```

- One other significant use of these modifier key methods is to test for which mouse button generated a particular mouse event on systems with two or three mouse buttons.
- By default, mouse events (such as mouse down and mouse drag) are generated regardless of which mouse button is used.
- However, Java events internally map left and middle mouse actions to meta and Control (Ctrl) modifier keys, respectively, so testing for the key tests for the mouse button's action.
- By testing for modifier keys, you can find out which mouse button was used and execute different behavior for those buttons than you would for the left button. Use an `if` statement to test each case, like this:

```
public boolean mouseDown(Event evt, int x, int y ) {  
    if (evt.metaDown())  
        // handle a right-click  
    else if (evt.controlDown())  
        // handle a middle-click  
    else // handle a regular click  
}
```

- Note that because this mapping from multiple mouse buttons to keyboard modifiers happens automatically, you don't have to do a lot of work to make sure your applets or applications work on different systems with different kinds of mouse devices.
- Because left-button or right-button mouse clicks map to modifier key events, you can use those actual modifier keys on systems with fewer mouse buttons to generate exactly the same results.
- So, for example, holding down the Ctrl key and clicking the mouse on Windows or holding the Control key on the Macintosh is the same as clicking the middle mouse button on a three-button mouse; holding down the Command (apple) key and clicking the mouse on the Mac is the same as clicking the right mouse button on a two- or three-button mouse.
- Consider, however, that the use of different mouse buttons or modifier keys may not be immediately obvious if your applet or application runs on a system with fewer buttons than you're used to working with.
- Consider restricting your interface to a single mouse button or to providing help or documentation to explain the use of your program in this case.

The awt Event Handler

- The default methods you've learned about today for handling basic events in applets are actually called by a generic event handler method called `handleEvent()`. The `handleEvent()` method is how the awt generically deals with events that occur between application components and events based on user input.
- In the default `handleEvent()` method, basic events are processed and the methods you learned about today are called. To handle events other than those mentioned here, you need to override `handleEvent()` in your own Java programs. The `handleEvent()` method looks like this:

```
public boolean handleEvent(Event evt) {
    ...
}
```

- To test for specific events, examine the `id` instance variable of the `Event` object that gets passed in to `handleEvent()`.
- The event ID is an integer, but fortunately the `Event` class defines a whole set of event IDs as class variables whose names you can test for in the body of `handleEvent()`.

- Because these class variables are integer constants, a `switch` statement works particularly well. For example, here's a simple `handleEvent()` method to print out debugging information about mouse events:

```
public boolean handleEvent(Event evt) {
    switch (evt.id) {
    case Event.MOUSE_DOWN:
        System.out.println("MouseDown: " +
            evt.x + "," + evt.y);
        return true;
    case Event.MOUSE_UP:
        System.out.println("MouseUp: " +
            evt.x + "," + evt.y);
        return true;
    case Event.MOUSE_MOVE:
        System.out.println("MouseMove: " +
            evt.x + "," + evt.y);
        return true;
    case Event.MOUSE_DRAG:
        System.out.println("MouseDown: " +
            evt.x + "," + evt.y);
        return true;
    default:
        return false;
    }
}
```

You can test for the following keyboard events:

- `Event.KEY_PRESS` is generated when a key is pressed (the same as the `keyDown()` method).
- `Event.KEY_RELEASE` is generated when a key is released.
- `Event.KEY_ACTION` and `Event.KEY_ACTION_RELEASE` are generated when an action key (a function key, an arrow key, Page Up, Page Down, or Home) is pressed or released.

You can test for these mouse events:

- `Event.MOUSE_DOWN` is generated when the mouse button is pressed (the same as the `mouseDown()` method).
- `Event.MOUSE_UP` is generated when the mouse button is released (the same as the `mouseUp()` method).
- `Event.MOUSE_MOVE` is generated when the mouse is moved (the same as the `mouseMove()` method).
- `Event.MOUSE_DRAG` is generated when the mouse is moved with the button pressed (the same as the `mouseDrag()` method).
- `Event.MOUSE_ENTER` is generated when the mouse enters the applet (or a component of that applet).
You can also use the `mouseEnter()` method.

- `Event.MOUSE_EXIT` is generated when the mouse exits the applet. You can also use the `mouseExit()` method.

In addition to these events, the `Event` class has a whole suite of methods for handling awt components. You'll learn more about these events tomorrow.

Note that if you override `handleEvent()` in your class, none of the default event-handling methods you learned about today will get called unless you explicitly call them in the body of `handleEvent()`, so be careful if you decide to do this. One way to get around this is to test for the event you're interested in, and if that event isn't it, call `super.handleEvent()` so that the superclass that defines `handleEvent()` can process things. Here's an example of how to do this:

```
public boolean handleEvent(Event evt) {
    if (evt.id == Event.MOUSE_DOWN) {
        // process the mouse down
        return true;
    } else {
        return super.handleEvent(evt);
    }
}
```

Also, note that like the individual methods for individual events, `handleEvent()` also returns a boolean. The value you return here is particularly important; if you pass handling of the event to another method, you must return `false`. If you handle the event in the body of this method, return `true`. If you pass the event up to a superclass, that method will return `true` or `false`; you don't have to yourself.