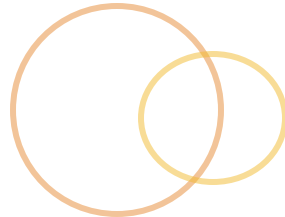
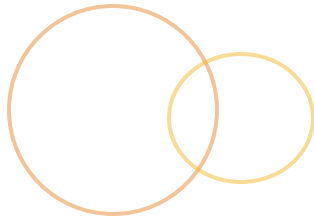
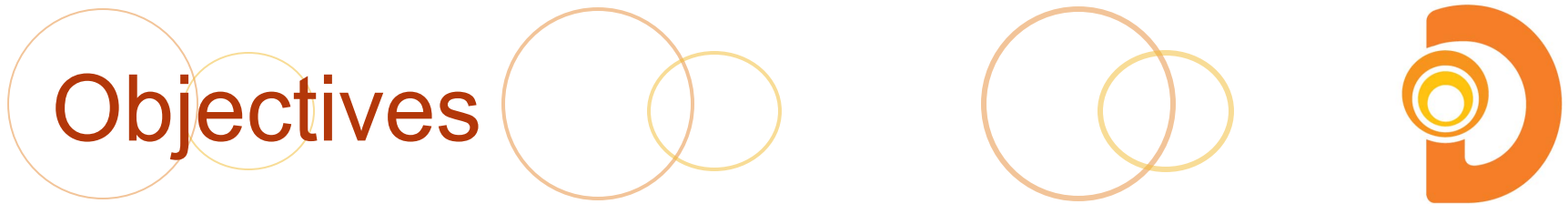


Review of the Java Platform



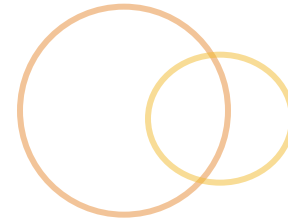


Objectives

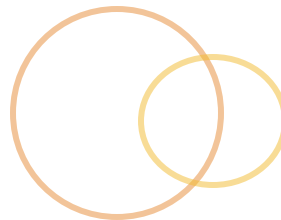
At the end of this module you should be able to:

- ② Understand Java
- ② Discuss why Java should be used and who owns it
- ② Talk about Java Classifications
- ② Use SDK & JRE
- ② Use the Java Programming Model
- ② Write, compile, and run Java Applications

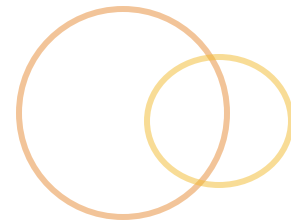
History of Java



Green is in!

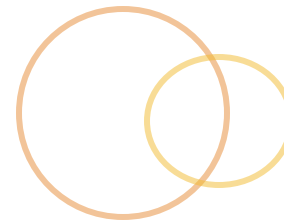


Just a Seed



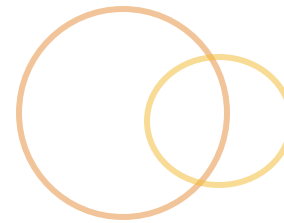
- ◎ Java started out as a research project
 - ◎ Research began in 1991 as the Green Project
 - ◎ Project was chartered to anticipate and plan for next wave of computing
 - ◎ “Green Team” determined consumer devices and computers would converge
 - ◎ Team focused on TV set-top boxes and interactive TV industries

Only a Sapling



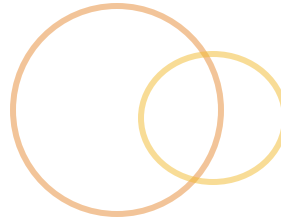
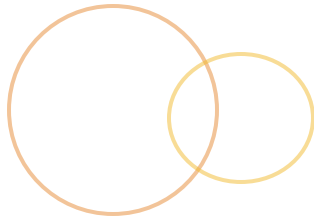
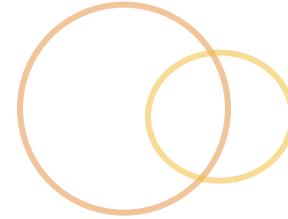
- Research efforts birthed a new language, OAK
 - Oak was renamed Java in 1994
 - Created by James Gosling - *“the father of Java”*
- Language was created with 5 main goals:
 - It should be object oriented
 - A single representation of a program could be executed on multiple operating systems
 - It should fully support network programming
 - It should execute code from remote sources securely
 - It should be easy to use

The Budding Tree



- ◎ Java was publicly released May 27, 1995
 - ◎ As a product, it was targeted at Internet development
 - ◎ In general, it was marketed as the language to add dynamic features to the web, a.k.a. Applets
 - ◎ Had early support from companies like Netscape Communications

What is Java?

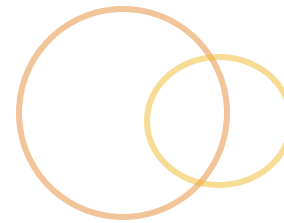


What is Java?



- ◎ Java is defined by two entities:
 - ◎ A platform (Java Runtime Environment - JRE)
 - ◎ A language (Java Software Development Kit - SDK)
- ◎ Java addresses less traditional concerns like
 - ◎ Security
 - ◎ Reusability
 - ◎ Transportability (platform independence)
 - ◎ Network capability
- ◎ Created by Sun Microsystems, now guided by Oracle

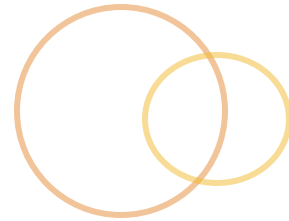
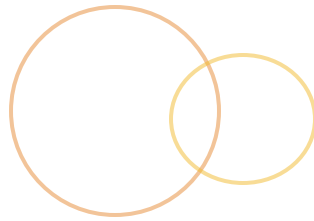
The Java Platform



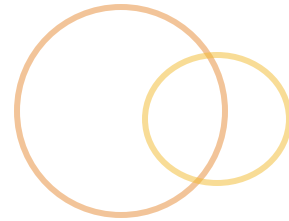
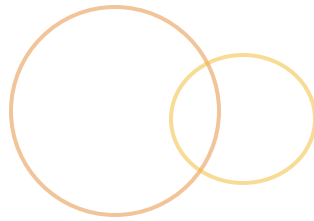
- ◎ The Java platform provides
 - ◎ The runtime environment
 - ◎ The necessary libraries (platform libraries)
- ◎ The Java platform software is NOT platform independent
 - ◎ Platform implementations exist for almost every operating system including Windows, OS X, Linux, Solaris, AIX, HP-UX, VMS, OS/2, OS/400, many embedded systems, and many more

Platform Editions

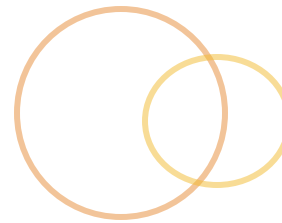
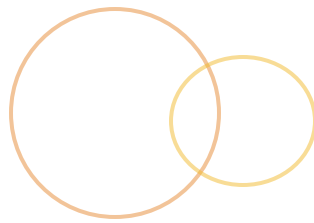
- ◎ Java has different platform editions
 - ◎ Java Standard Edition (Java SE)
 - ◎ Java Enterprise Edition (Java EE)
 - ◎ Java Micro Edition (Java ME)
- ◎ Editions defined in terms of JVM and platform libraries
 - ◎ Each platform has its own set of “libraries”
 - ◎ All editions rely on a Java Runtime Environment and Java Virtual Machine



- Complete environment for application execution
 - Standalone server applications
 - Standalone client applications
 - Standalone client-server applications
 - Applets
 - Mobile applications
 - Web-start applications -- rich applications deployed via Web
- Considered 'core' to all editions

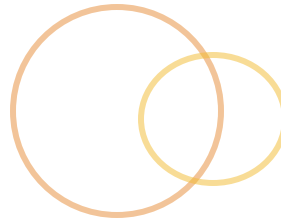
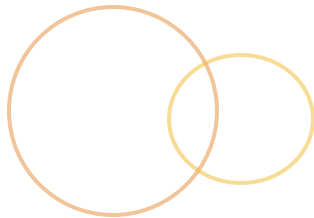


- ⦿ Extension of Java SE - uses Java SE run-time environment
 - ⦿ Adds libraries, frameworks, and container concept
- ⦿ Targeted at enterprise applications -- applications that span all areas of an enterprise
 - ⦿ From customer to back-office
 - ⦿ From web to legacy
- ⦿ Enables distributed multi-tier solutions

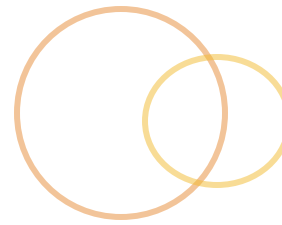


- Targeted at consumer and embedded market; constrained devices
- Defined variations for major categories of device physical capability
 - Connected Device Configuration (CDC)
 - Connected Limited Device Configuration (CLDC)

Current State of Java

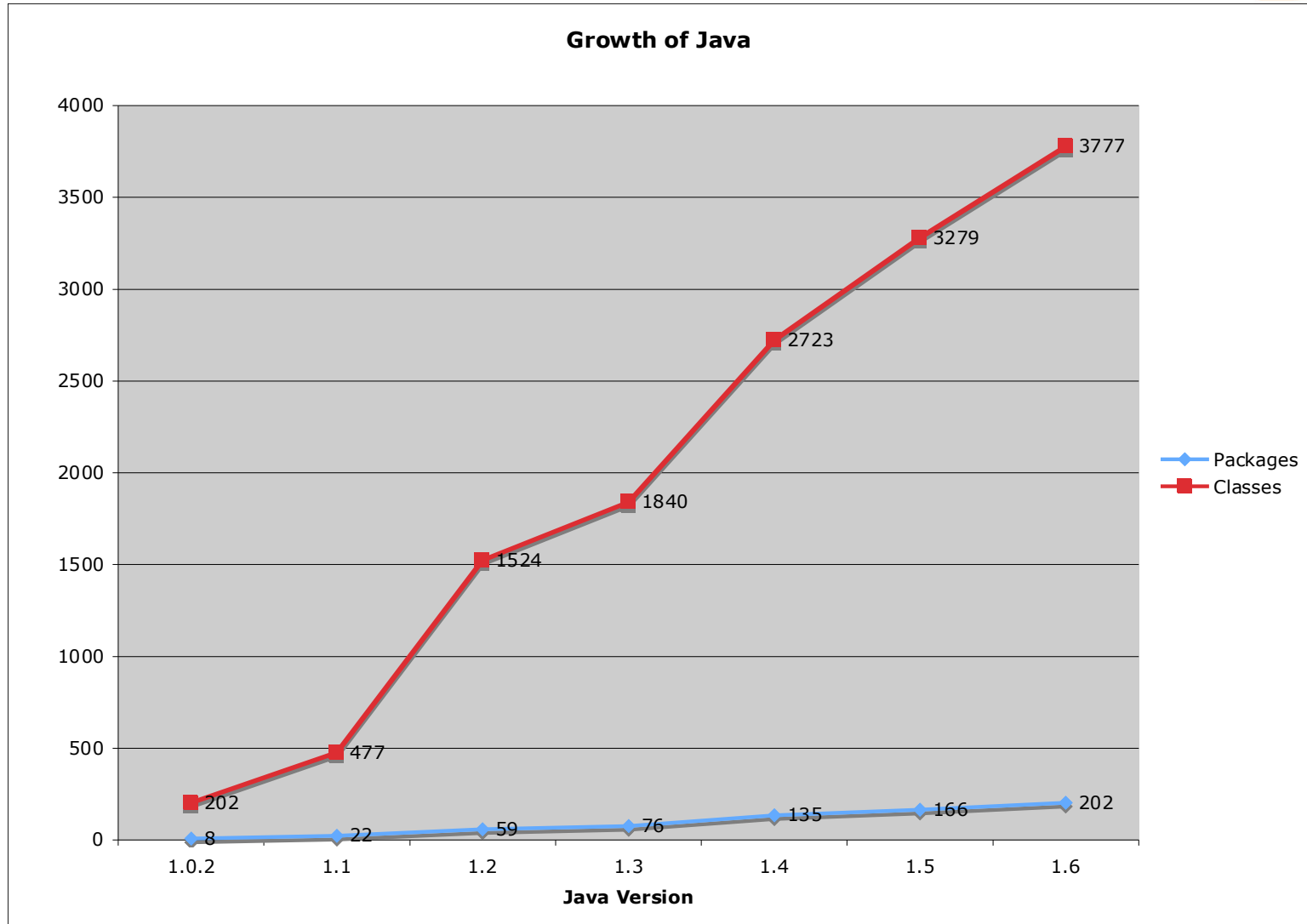


A Growing Forest

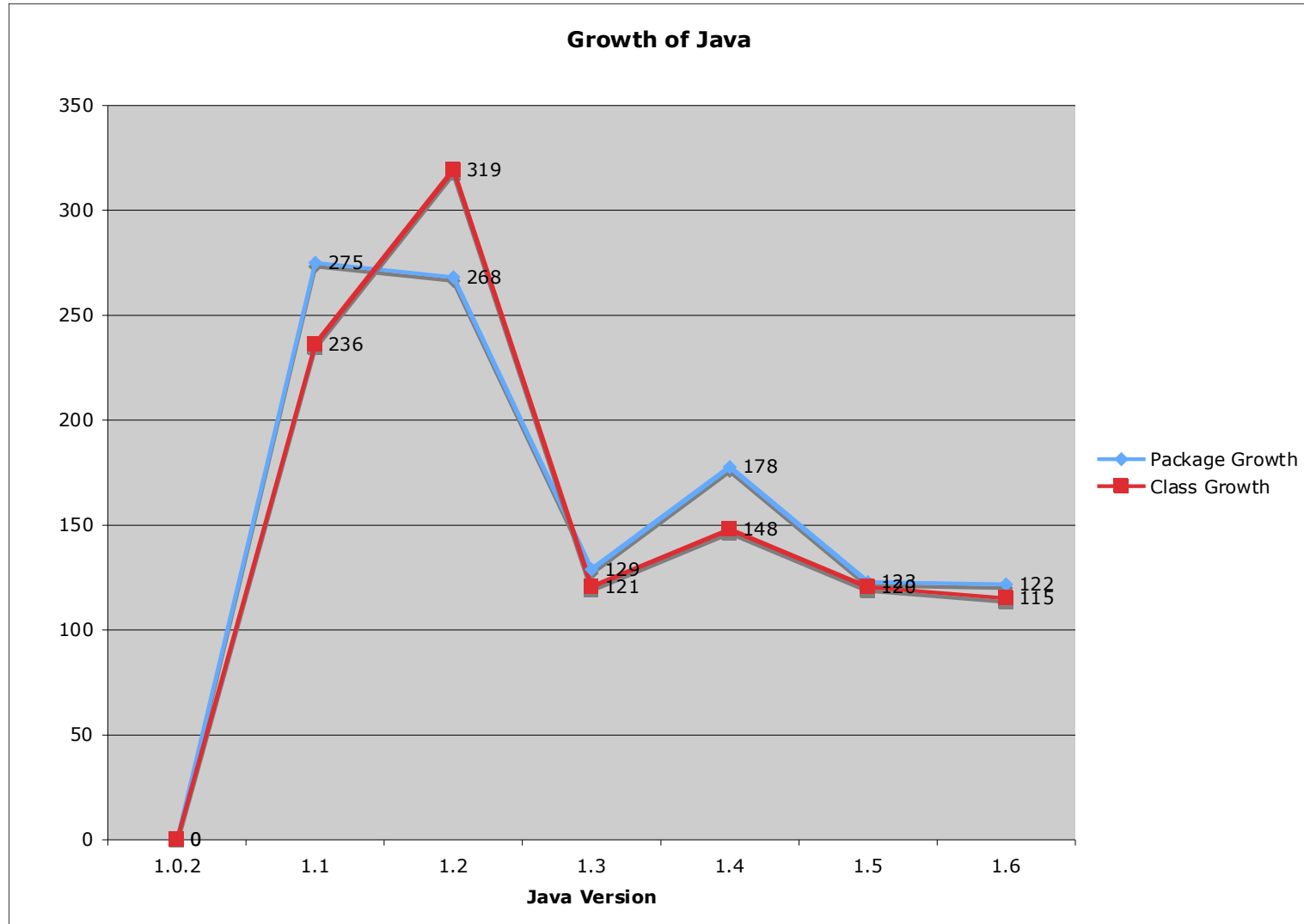


- First released in 1995
- Version 1.2 branded “Java2” or “J2SE 1.2” – Java2 Standard Edition
- Version 1.5 rebranded “Java 5” or Java SE 5
 - At same time, J2EE 1.4 rebranded to Java EE 5 with release of EE 1.5
- Current versions:
 - Java SE is Java 8 (a.k.a 1.8)
 - Java EE is Java EE 7 (a.k.a 1.7)

Breadth of Java Across Versions



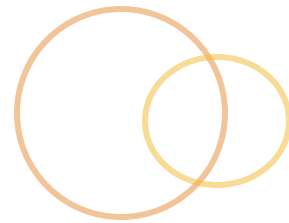
Growth of Java Across Versions



Java Standard Edition



Java SE Platform



- Represents the historic Java platform
- Considered the core Java platform
 - Used for browser plugins to standalone Java applications
 - Extended to support enterprise application development (Java EE)
 - Constrained to support micro application development (Java ME)
- Typically discussed in terms of its
 - Runtime environment (JRE)
 - Development environment (JDK)

Java SE Platform

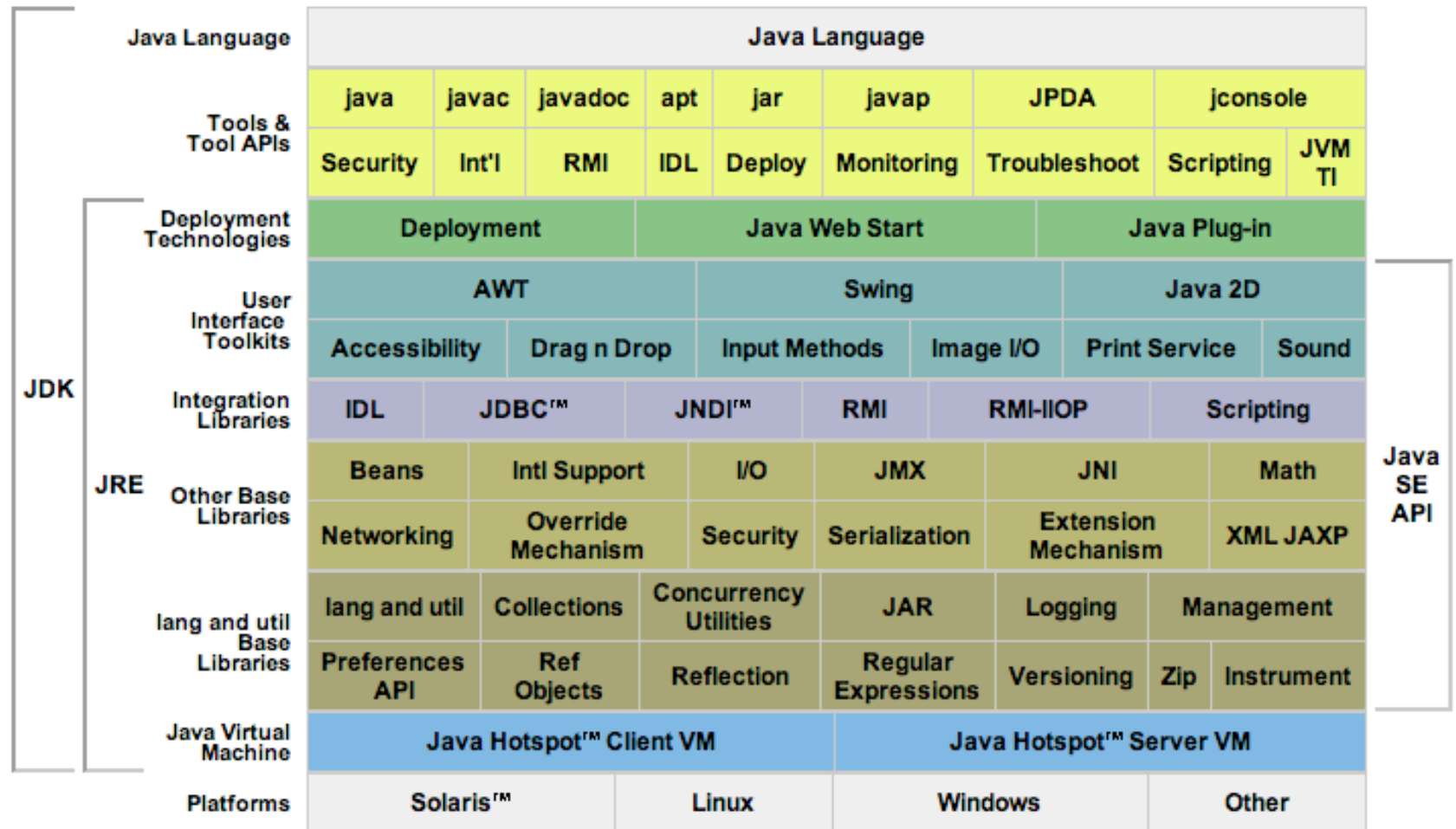


Image location: <http://java.sun.com/javase/6/docs/>

Java SE Runtime Environment



- Considered the execution platform
- Consists of two primary facilities
 - Java Virtual Machine (JVM)
 - Java SE Application Programming Interfaces (API)
- When put together, they are considered the JRE
 - JRE implementation is operating system specific
 - JRE has consistent behaviors and capabilities across operating systems
 - JRE is the only necessary piece required to run a Java application

Java SE Platform [JVM]

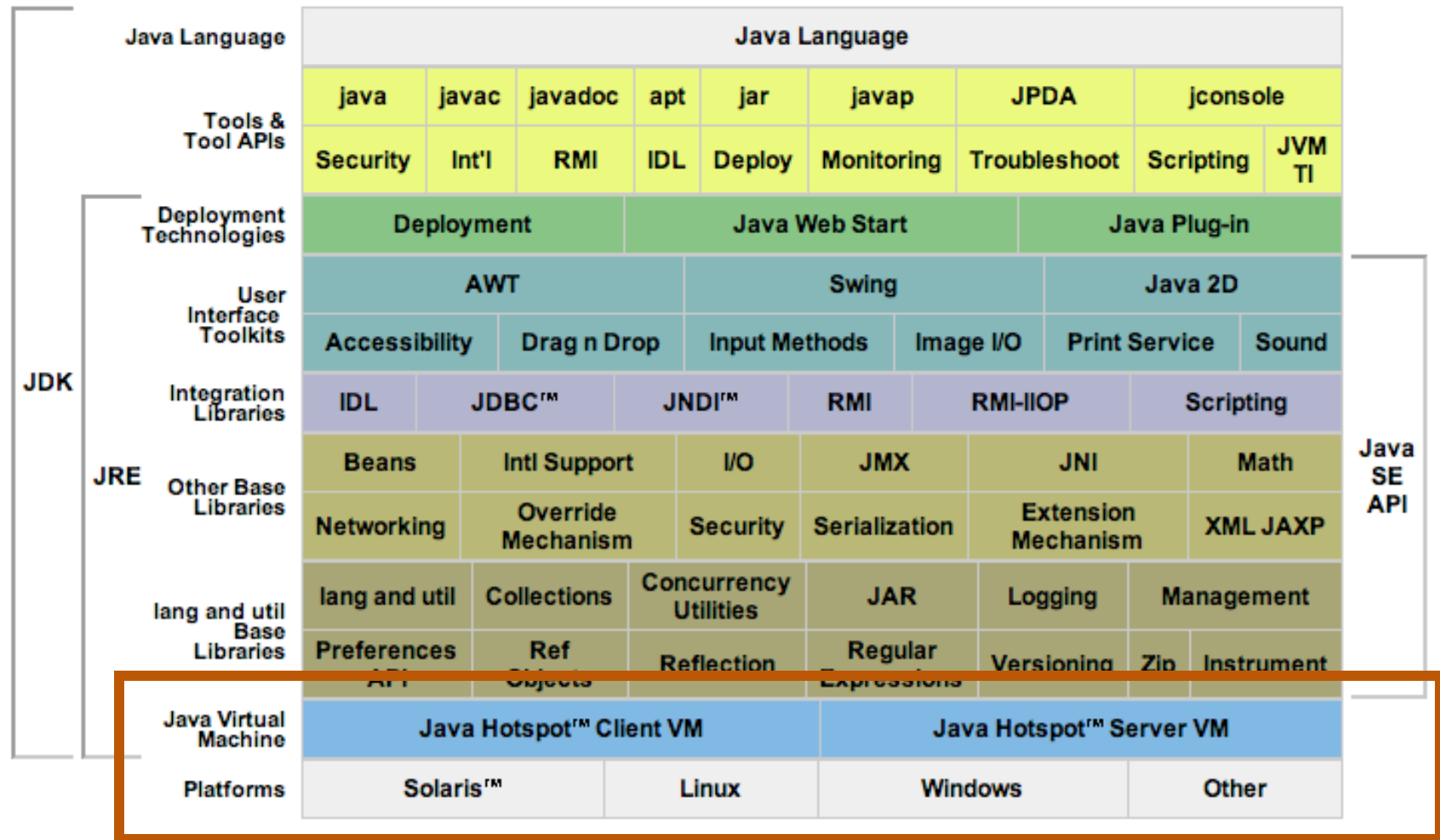


Image location: <http://java.sun.com/javase/6/docs/>

Java Virtual Machine



- Stand-alone OS native application
 - Executes bytecode
 - Bytecode represents compiled Java source code
 - JVM is operating system dependent
- JVM acts as facilitator between a Java application and the OS
 - Isolates application from OS quirks
 - Provides consistency across OS to application
 - Contains a host of rich execution facilities
- Specification driven

Java Virtual Machine [cont.]



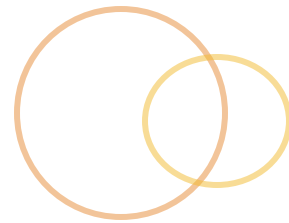
◉ JVM variations

- ◉ *Interpreters* – first form of Java: pure interpretation
- ◉ *Compilers* - native code; fast, but platform-dependent binary
- ◉ *Just-In-Time Compilers (JIT)* – Current form of Java: partial interpretation, partial compilation at runtime

◉ JVM Implementations

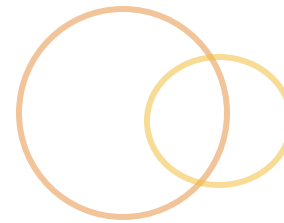
- ◉ JVM functionality is defined by a specification
- ◉ Different vendors have different implementations
- ◉ May have different execution modes and tuning characteristics

JVM Facilities



- ◉ JVM provides:
 - ◉ Platform-independent execution
 - ◉ Dynamic binding
 - ◉ Thread management
 - ◉ Automatic memory management
 - ◉ Security model

Java SE API



- Typically discussed in terms of libraries (APIs)
 - APIs are presented as “packages”
 - Each package is a logical grouping of related functionality
- Libraries broken into four categories:
 - Language
 - Base
 - Integration
 - UI toolkits
- Platform may be “extended” through additional packages
- Community driven

Java SE Platform [API]

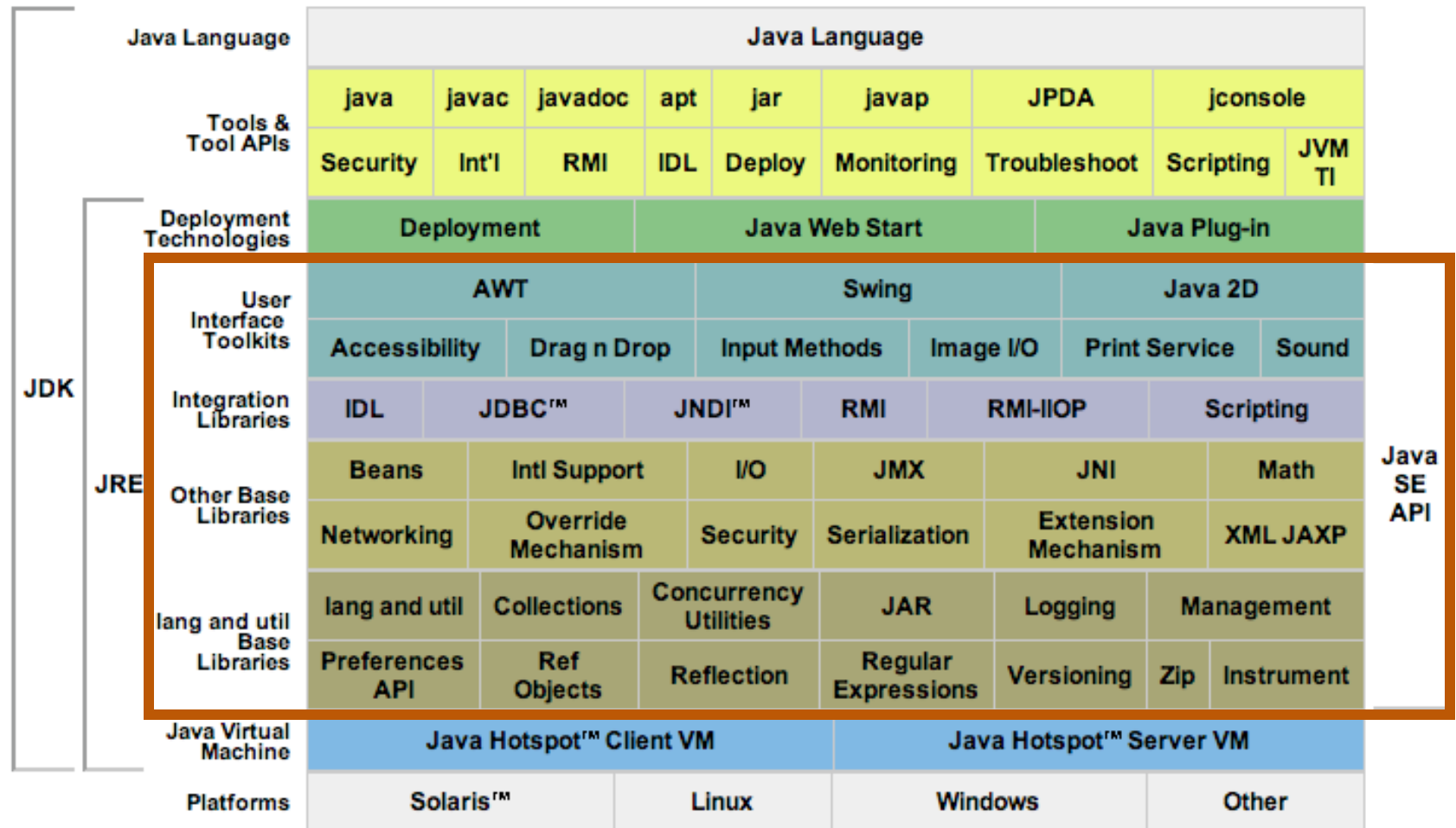


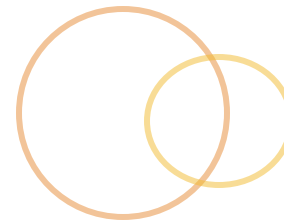
Image location: <http://java.sun.com/javase/6/docs/>

Java Language Packages



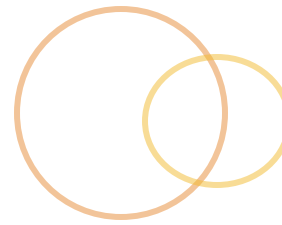
- Provide implementation of language characteristics and functionality
- Governed by the Java Language Specification
- Available in all platforms
- Typically found in `java.lang` packages like:
 - `java.lang`
 - `java.lang.annotation`
 - `java.lang.ref`
 - `java.lang.reflect`

Base Packages



- Considered foundational to the SE platform
- Normally found in `java.` package structure, like:
 - `java.io`
 - `java.net`
 - `java.util`
- Subset of base packages found in Java ME

Base: Input / Output



- Provides platform-independent I/O mechanism
 - Supported by abstraction of file system
 - OS specifics handled by native implementation
- Two types of I/O
 - Synchronous I/O - `java.io`
 - Follows stream-based model
 - Supports text and binary
 - High performance I/O - `java.nio`
 - Follows channel-based model
 - Asynchronous
 - Supports buffers

Base: Network Programming



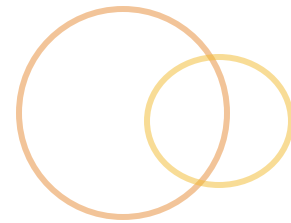
- ◎ Rich support for networked applications
 - ◎ Found in `java.net`
 - ◎ Underlying communication handled by OS
- ◎ Supports transport layer communication
 - ◎ TCP - sockets and server sockets
 - ◎ UDP - packets and sockets
- ◎ Support for application-layer programming
 - ◎ Through `java.net.URL` and `java.net.URLConnection`
 - ◎ Includes support for things like:
 - ◎ Http
 - ◎ Mailto
 - ◎ FTP

Base: Data Structures



- Collections API provides standard data structures
- Lists, sets, maps, queues, etc.
- Thread-safe and non-thread-safe implementations
- Built-in and extensible sorting, selection, and ordering facilities

Integration Packages



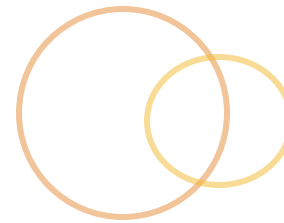
- ⦿ Contain libraries and functionality to integrate with other systems
- ⦿ Implemented using a layered approach
 - ⦿ Provides Write-Once-Run-Anywhere (WORA) integration capabilities
 - ⦿ Abstract the application from system specifics
- ⦿ Includes things like:
 - ⦿ `java.sql`
 - ⦿ `java.rmi`
 - ⦿ `javax.xml`

Integration: Database Programming



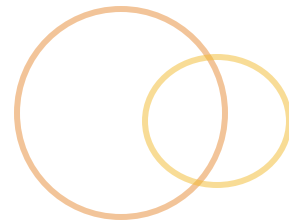
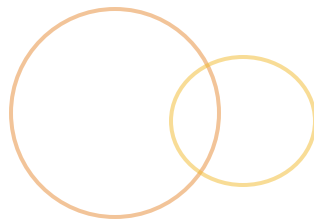
- ◎ Java Database Connectivity (JDBC)
 - ◎ WORA for databases
 - ◎ JDBC provides a set of database independent APIs
 - ◎ DB specific interactions provided by JDBC-compliant driver
 - ◎ Supports connections to multiple databases at a given time
- ◎ Found in two packages:
 - ◎ `java.sql`
 - ◎ `javax.sql`
- ◎ Capabilities leveraged by Java EE

Integration: RMI



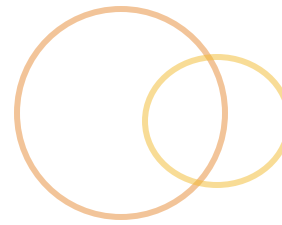
- ◉ Stands for remote method invocation
- ◉ Java-specific distributed computing mechanism
- ◉ Introduced in JDK 1.1
- ◉ Built into the Java platform - `java.rmi`
- ◉ Distributed computing platform for:
 - ◉ Distributed object-oriented computing
 - ◉ Enterprise Java Beans

UI Toolkits



- ⦿ User Interface development supported through:
 - ⦿ Abstract Windowing Toolkit
 - ⦿ Basic; least-common-denominator widget set
 - ⦿ Relies on *native-peers*
 - ⦿ Platform-specific look and feel
 - ⦿ Swing / Java Foundation Classes
 - ⦿ Advanced; full-featured
 - ⦿ Written in Java
 - ⦿ Pluggable look and feel with option for consistent platform-independent look and feel
- ⦿ Provides WORA for graphical-based applications

Platform Extensions



- Considered extensions to the platform
 - Not necessarily considered “core” facility of platform
 - Typically governed by specification falling outside “platform specification”
 - Usually bundled with platform, but could be third-party
- Typically have a `javax` package structure like:
 - `javax.naming`
 - `javax.swing`
 - `javax.transaction`
- Many `javax` packages are now in the core
 - Inconvenient to rename once released & in regular use

Types of Java Applications



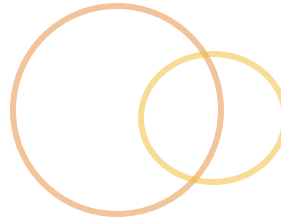
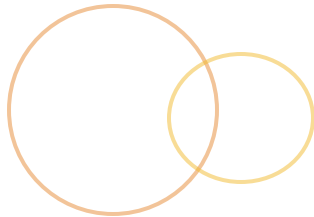
- ◎ Java SE is targeted at creating “classic” applications
 - ◎ Few constraints on class; any class can be an application
 - ◎ JVM executes a lifecycle method
 - ◎ Lifecycle method must have specific signature:

```
public static void main(String [ ] args)
{ ... }
```
- ◎ Classic applications include:
 - ◎ Standalone client
 - ◎ Client-server
 - ◎ Server
 - ◎ Distributed (peer-to-peer)

Types of Java Applications [cont.]

- ◎ Java SE also supports some web-based application approaches
- ◎ Two types of web-based applications
 - ◎ Applets (embedded in web pages)
 - ◎ Web-start applications (installed on client if desired and updated from server if changes made)
- ◎ Both have similar characteristics in terms of:
 - ◎ Deployment
 - ◎ Security
 - ◎ Execution

Java SE Development



Java SE Development



- Application development provided through Java Development Kit (JDK)
- JDK contains:
 - Java SE JRE
 - A set of development tools
- All development tools are command-line
- Rich development environment provided through Integrated Development Environments (IDEs)

Java SE Platform [JDK]

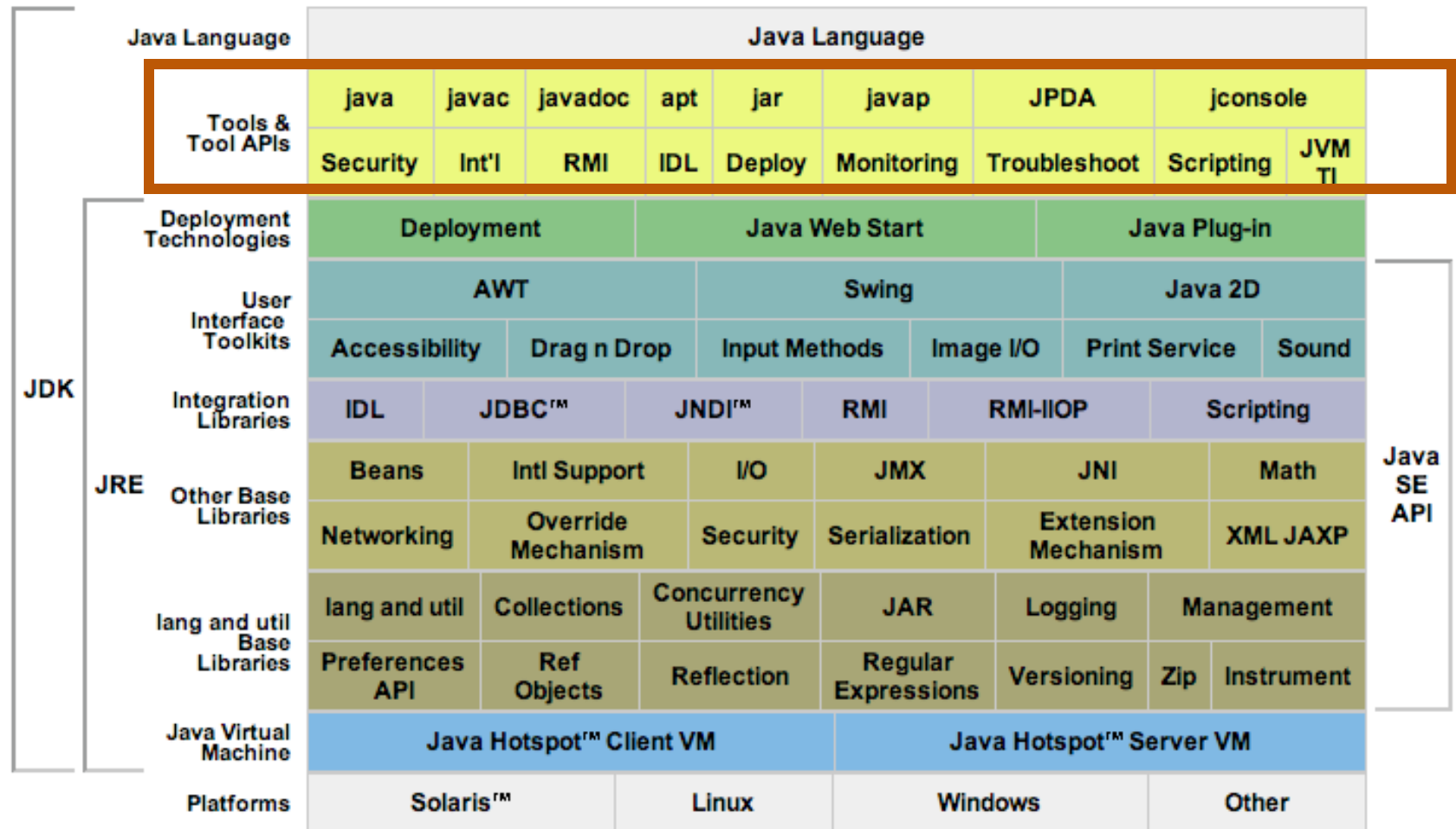


Image location: <http://java.sun.com/javase/6/docs/>

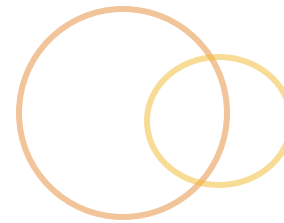
Core Java Development Concepts



Java language source code defined in text files:

- Source files provide:
 - Definition of entities and rules
 - Description of how entities interact
- Source files have basic requirements:
 - Filenames are case and white-space sensitive
 - File extension must be `.java`
- Source files become executable after compilation
 - Executable files contain bytecode
 - File extension is `.class`
 - At least one bytecode file generated per source file
 - Bytecode files are platform independent

Java Compiler



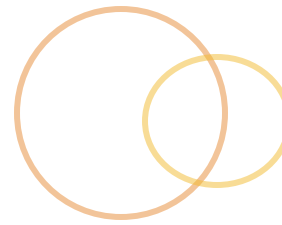
- ⦿ Java compiler provided as part of JDK
- ⦿ Written as Java program
 - ⦿ Invoked on the command-line: `javac`
 - ⦿ Relies on a classpath
- ⦿ Uses multi-pass algorithm
 - ⦿ Basic syntax checking
 - ⦿ Type verification / validation checking
 - ⦿ Exception handling
 - ⦿ Identifies and notifies errors (and line numbers)
- ⦿ Generates Java Virtual Machine compliant bytecode

Java Application Launcher



- Application launcher provided as part of JRE
- Used to start a standalone Java application
 - Invoked on the command line: `java`
 - Starts a Java Runtime Environment
 - Loads platform libraries
 - Loads and starts the application
- Has many configurable options
 - Classpath
 - Memory management algorithm
 - Memory size
 - Remote management

Other JDK Tools



- ◎ Java Documentation Generator
 - ◎ Command-line documentation tool: `javadoc`
 - ◎ Generates HTML-based documentation from source code
 - ◎ Useful for creating developer-oriented documentation
- ◎ Java Debugger
 - ◎ Command-line debugger: `jdb`
- ◎ Third party IDEs combine editor, documentation, debugging, refactoring, version control, and other tools:
 - ◎ NetBeans (Sun/Oracle), Eclipse (IBM), JDeveloper (Oracle), and many others

Java Programming Model



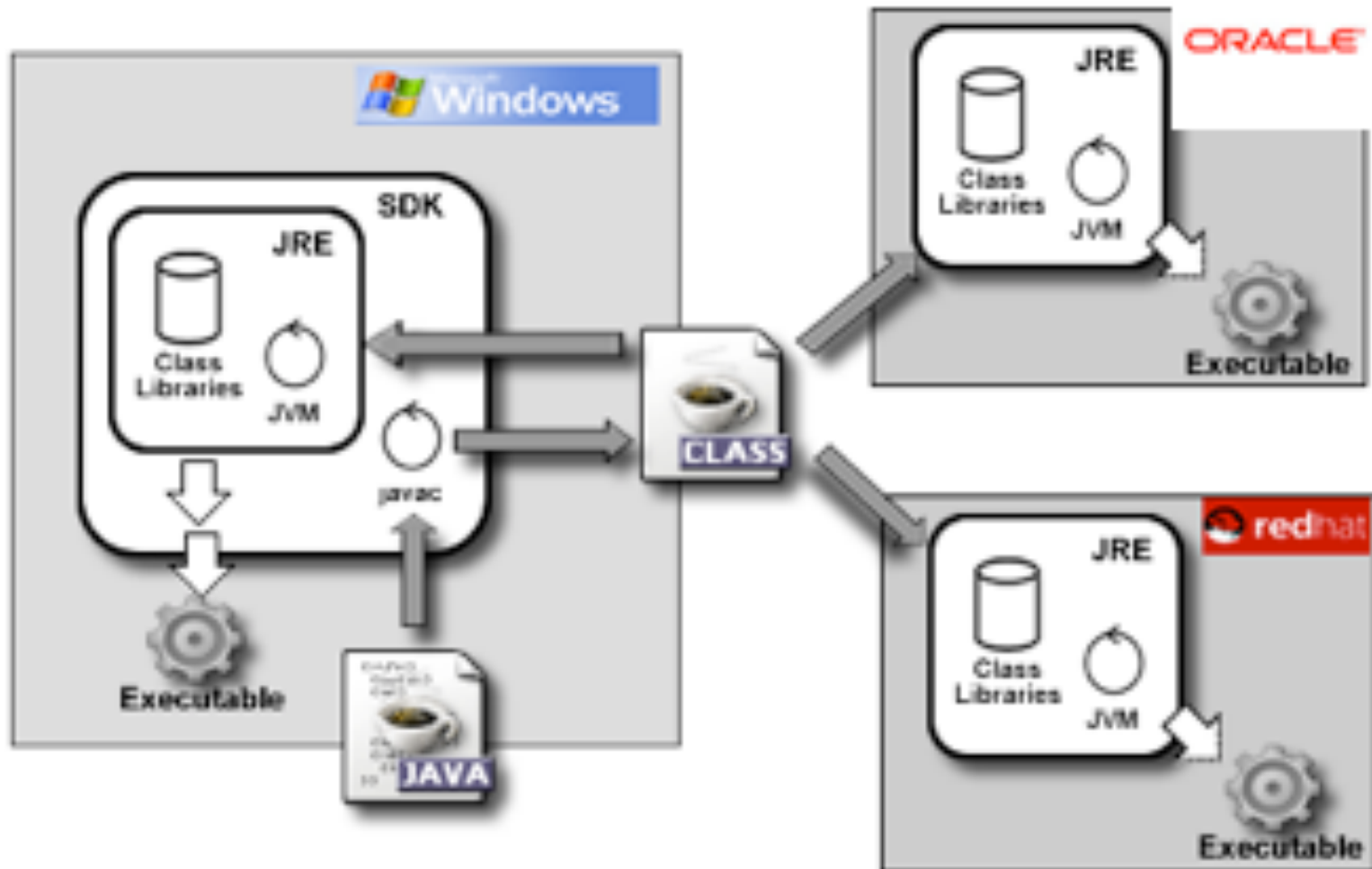
- Create source code
 - Stored as text file
 - Has extension .java
- Compile source code
 - Utilize java compiler - javac
 - Performs syntax and language validation
 - Generates platform independent bytecode
 - Stored in a .class file

Java Programming Model

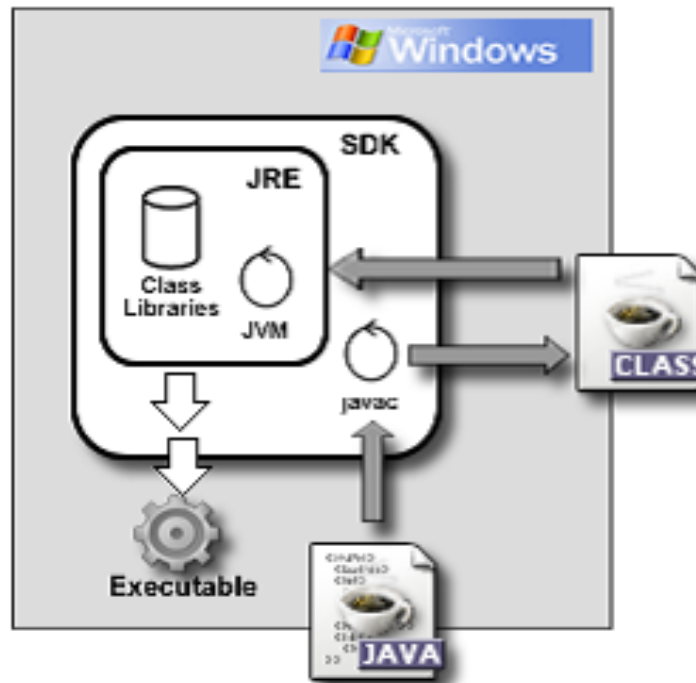


- Distribute .class files
 - On the web (for applets)
 - On the server (for enterprise applications)
 - On the client (for applications)
- Execute the application
 - Use the Java Runtime Environment (JRE)
 - JRE utilizes a Java Virtual Machine (JVM)
 - JVM loads class files and executes them
- Or let the IDE handle compilation/execution

Java Programming Model



Standalone Java Application



Compiling and Executing Java Applications



Developing

HelloWorld.java

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



```
> javac HelloWorld.java
```



HelloWorld.class

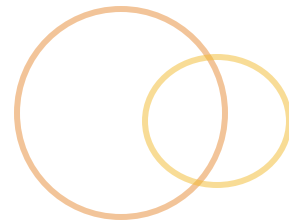
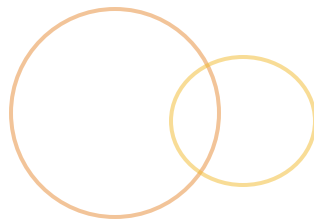
Executing

```
> java HelloWorld
```



Hello World!

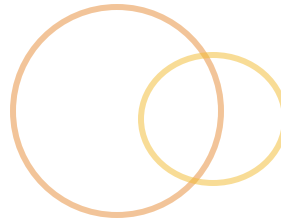
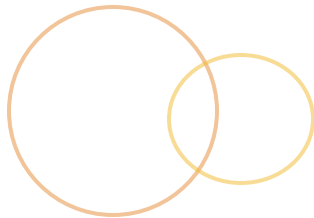
Summary



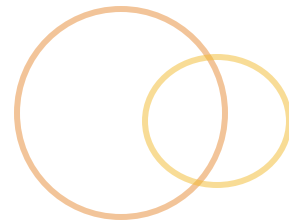
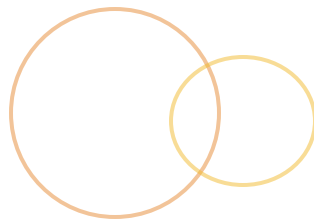
We covered:

- ② Understand Java
- ② Discuss why Java should be used and who owns it
- ② Talk about Java Classifications
- ② Use SDK & JRE
- ② Use the Java Programming Model
- ② Write, compile, and run Java Applications

Creating and Running Java Programs



Objectives



At the end of this module you should be able to:

- Understand the layout of a Java class
- Create your first Java class
- Use the System class to print a string to the console
- Open and configure Eclipse to run labs in a Java Project
- Run the program and examine output

Class Structure



- ⦿ A Java class is program or a portion of a program
- ⦿ All Java programs are contained within Java classes
- ⦿ There are three types of declarations that can exist outside of a class declaration
 - ⦿ Package declaration
 - ⦿ Import statement
 - ⦿ Comment

The Most Basic Java Class



```
class MyFirstClass  
{  
}
```

- ⦿ All Java Classes begin with a class declaration
 - ⦿ Other types of classes include interfaces and enums
- ⦿ Classes contain variables and methods

Java Methods



- ⦿ Methods in Java are identified by a method signature
- ⦿ Method signatures always consist of a return type, a name, and arguments
- ⦿ When calling a method you must supply the exact number and type of arguments
- ⦿ All methods are defined by enclosing curly braces
- ⦿ All Java programs begin with the main method:

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

Java Variables



- ⦿ Identifiers are used to name variables
- ⦿ Variables can exist inside or outside of methods
- ⦿ Curly braces define the scope of variables
 - ⦿ Scope will be discussed in depth in a later module
- ⦿ Variables must conform to the Java Specification and must not use restricted words
- ⦿ Java is a case-sensitive language

```
int i = 0;  
String s = "Hello World;  
char c = 'H';  
float f = 3.145;
```

Reserved Keywords Cannot be Used as Variable Names



abstract	continue	for	new	switch
assert	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const*	float	native	super	while
true	false	null		

*goto and const are reserved but not used

true, false, and null are reserved literal names

Java Uses the Dot Notation

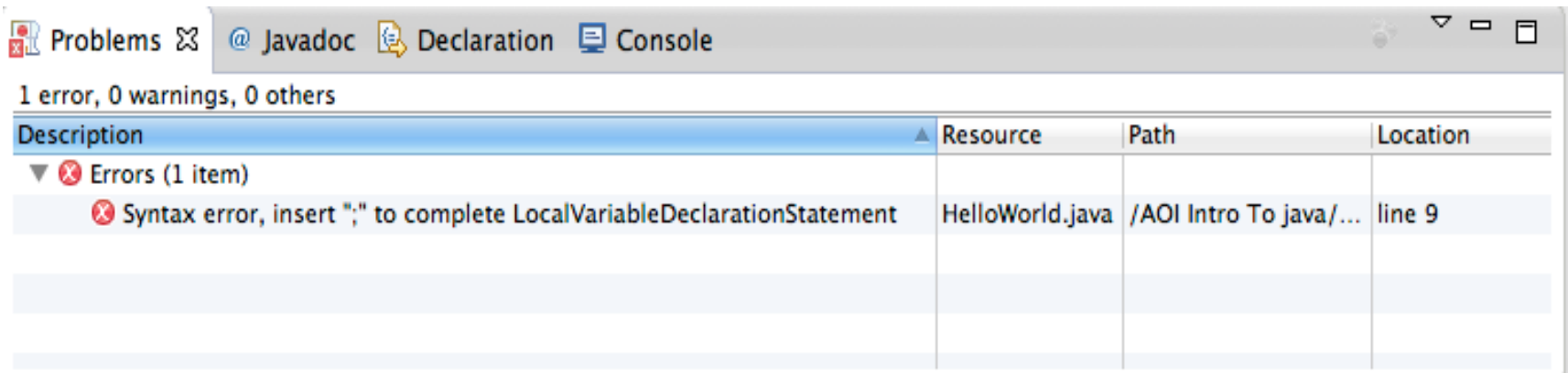


- Java uses the “.” notation to traverse hierarchy
- In order to use the output stream in our Hello World program we will need to use a special variable in the system class to call a method that will print out our string
- *`System.out.println("Hello World");`*

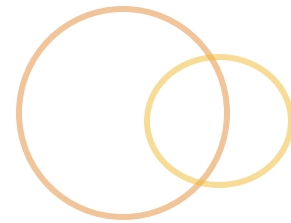
Java Terminates Declarations with a Semicolon;



- ☉ Java uses the semicolon to terminate declarations
- ☉ An error will be reported at compile time if semicolons are not used or are misplaced



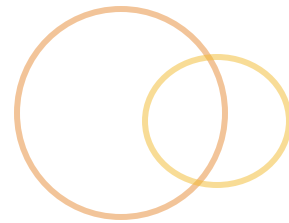
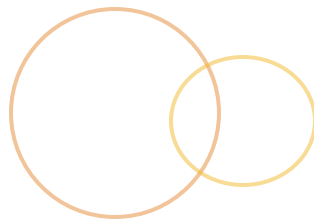
System.out.println();



● The Java Hello World

```
System.out.println("Hello World");
```

- System is a class
 - Classes are always capitalized
- out is a variable representing a PrintStream Object in the System class
 - Variables start with lower case
- println() is a method in the PrintStream class,
 - Methods also start with lower case

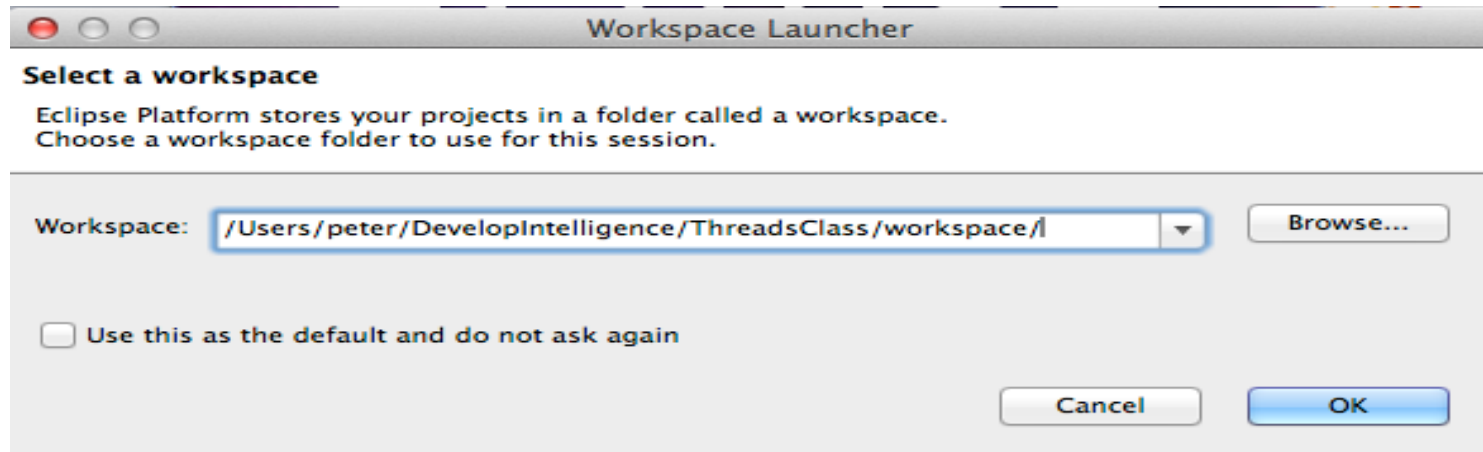


- Eclipse is an Integrated Development Environment
- Eclipse is free and open source
- There are many plugins that can be added to Eclipse to make programming easier
- We will use Eclipse to do all of our labs

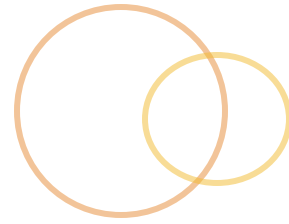
Creating a Project for our Programs



- Eclipse uses a directory structure called a workspace to organize all of its project files
- Start Eclipse and point it to the correct directory
- When Eclipse starts it will ask you what directory it should keep all of its workspace files in



Creating a Project

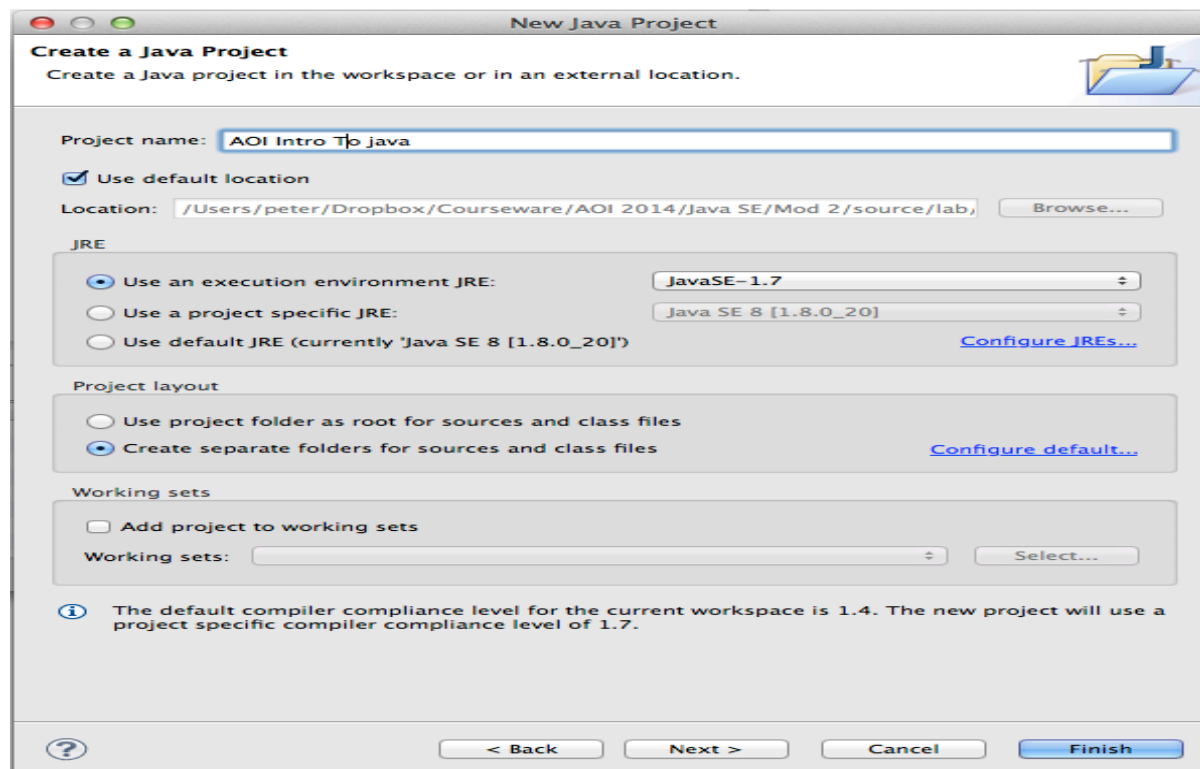


- All Programs in Eclipse run in a project
- There are many different types of projects available
- Projects have specific properties
 - Java version is an example of a property
- Each project defines a layout and a set of tools
- Layouts are customizable

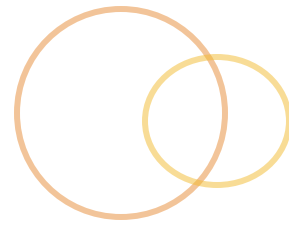
Declaring a Project



- Right click in the empty project explorer pane and select new project. Alternatively, you can select File → New → Project.

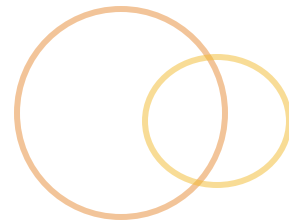


Boolean Data Types



- boolean data is either `true` or `false`
- In Java, numeric and other variables are not boolean, and cannot be interpreted as such
 - (In C, C++, JavaScript and others, zero is false, any defined non-zero value is true)
- A boolean can only have the value `true` or `false`

Perspectives



- A specific layout in Eclipse is called a perspective
- Perspectives are tailored to project types
- We will use the Java perspective



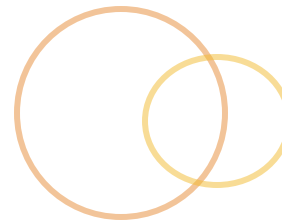
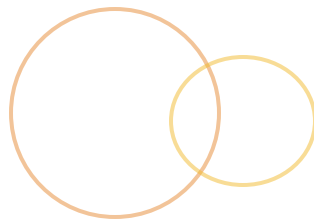
Integral Numeric Data Types



- Integral values are signed
 - char type has numeric properties, but is unsigned
- Integral values do not contain a decimal point

Type	Bytes	Minimum Value	Maximum Value
byte	1	-128	127
short	2	-32768	32767
int	4	-2147483648	2147483647
long	8	-9223372036854775808	9223372036854775807

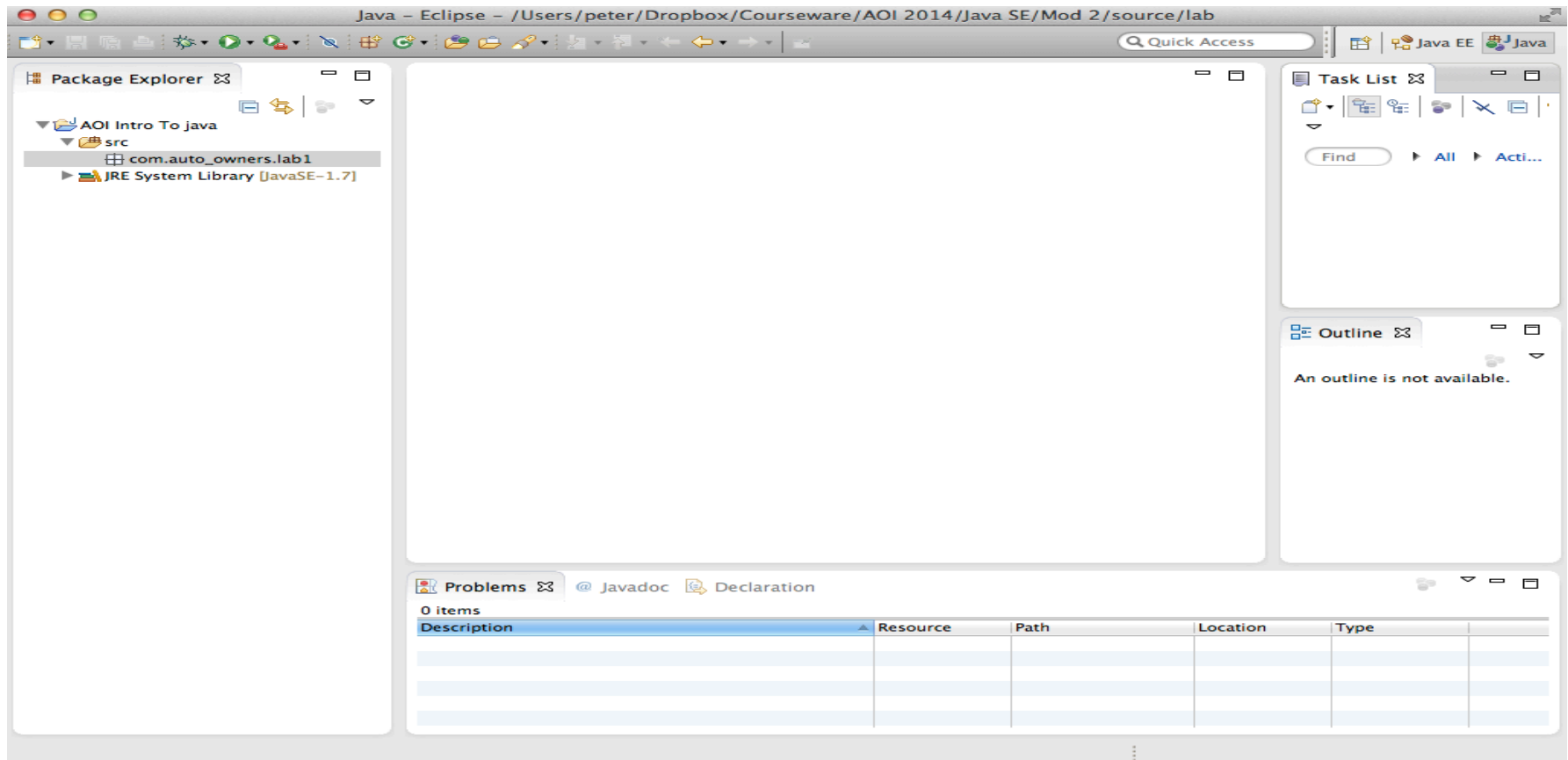
Packages



- The next step is to create a package to manage your source code.
 - We will talk more about packages. For now let's say that it is a mechanism to uniquely distinguish your HelloWorld from anyone else's HelloWorld.
 - We will use `com.auto_owners.lab1`
 - To create a package, right click on the project in the left pane and select: New → Package

We Are Now Ready To Say Hello

- Our Workspace is set up and we can now create a program



Creating Our First Class



- ⦿ Right click on the package and select New →
 - ⦿ Notice all of the possibilities under New
 - ⦿ If you select Other you will see even more possibilities, which will become clearer by the time we are finished.
- ⦿ Select New → Class. A new form will appear. For now let's just worry about the class name, which will be HelloWorld.

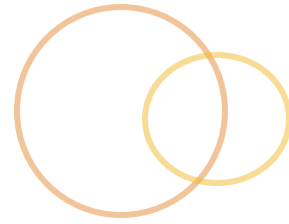
Fill In the Main Method



- ⦿ All Java programs start in the main method
- ⦿ Eclipse is nice enough to generate a class stub
- ⦿ Notice the structure of the class

```
package com.auto_owners.lab1;  
public class HelloWorld  
{  
    public static void main(String[ ] args)  
    {  
        System.out.println("Hello World");  
    }  
}
```

Run the Example

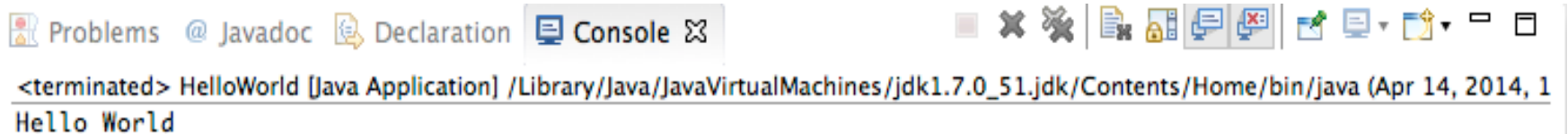


- Once the program is written, use the green dot with the white arrow on the top icon bar to run the program:



Program Results

- View the output in the console at the bottom of Eclipse

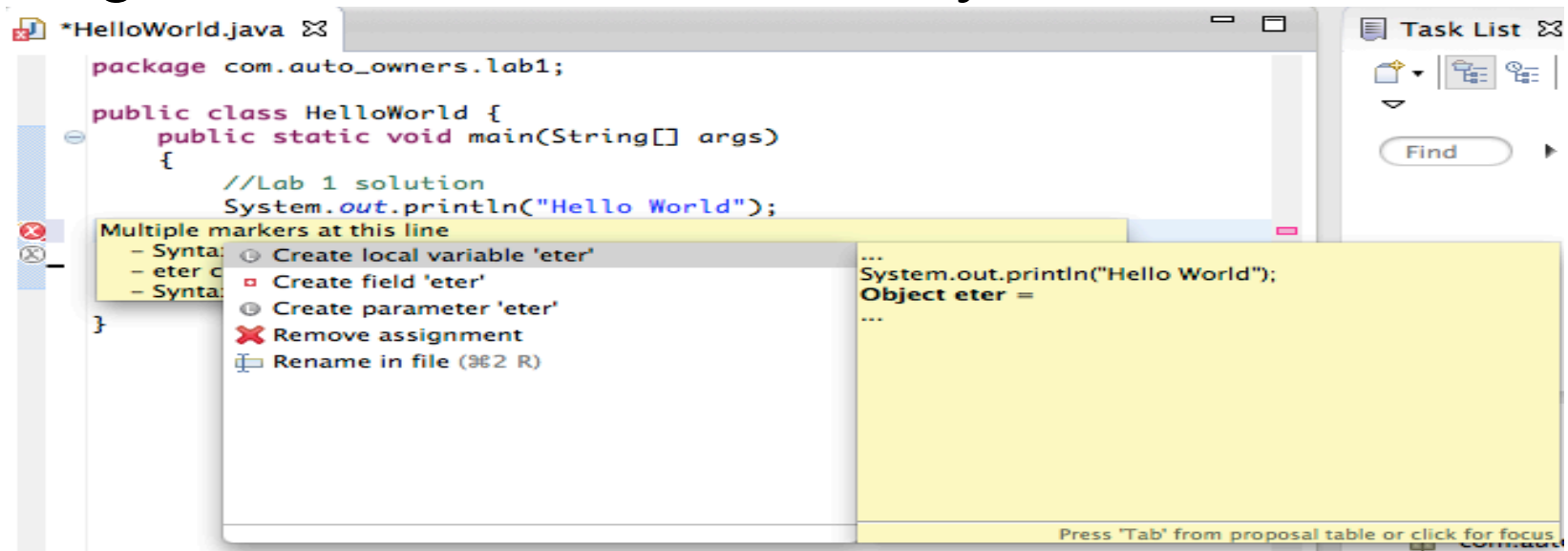


The screenshot shows the Eclipse IDE's console window. The title bar includes tabs for 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console output displays the text '<terminated> HelloWorld [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home/bin/java (Apr 14, 2014, 1' followed by a new line with 'Hello World'.

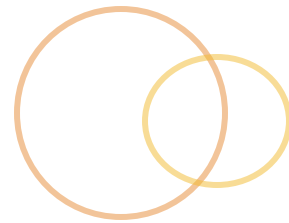
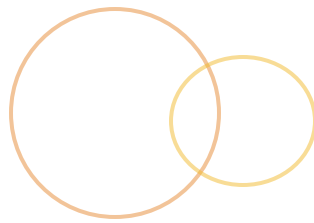
```
<terminated> HelloWorld [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home/bin/java (Apr 14, 2014, 1
Hello World
```

Errors

- If there are errors, they will show up as red underlines.
- If you click on the underlined code, it will give you options to fix the error or you may have to figure out the error and fix it yourself.

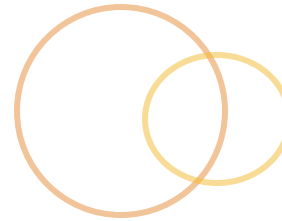
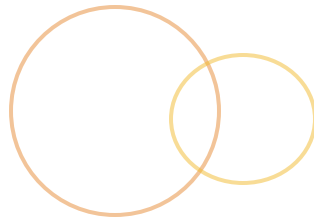


Summary



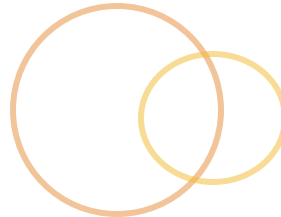
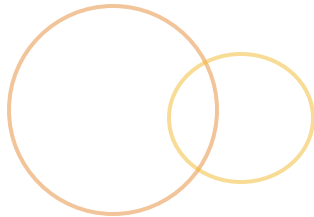
We covered:

- 🕒 Creating a simple Java class
- 🕒 Class structure
- 🕒 Eclipse
- 🕒 Running a program

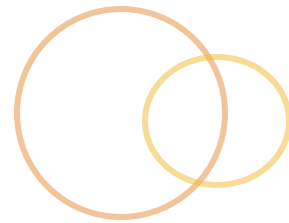
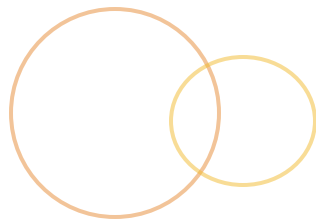


● Create and compile **HelloWorld!**

Objects And Classes (Part 1)



Objectives



- At the end of this module, you should be able to:
 - Use the `new` operator to create objects
 - Describe how reference variables work
 - Use instance variables and methods
 - Discuss the use of constructors

Creating Objects



- ⦿ Every OOP language must have a mechanism for creating objects from the class definitions
- ⦿ Java uses the instantiation mechanism found in other OOP languages -- the `new` operator
- ⦿ There is one normal way to create an object in Java - by using the `new` operator
- ⦿ The `new` operator is used in conjunction with a *constructor* to create (*instantiate*) an object
- ⦿ The virtual machine is responsible for creating the memory associated with the object and initializes the memory through a constructor

Creating a BankApp Object



```
public class BankApp {  
    public static void main(String [] args) {  
        // create the BankApp object  
        BankApp thisApp = new BankApp() ;  
    }  
}
```

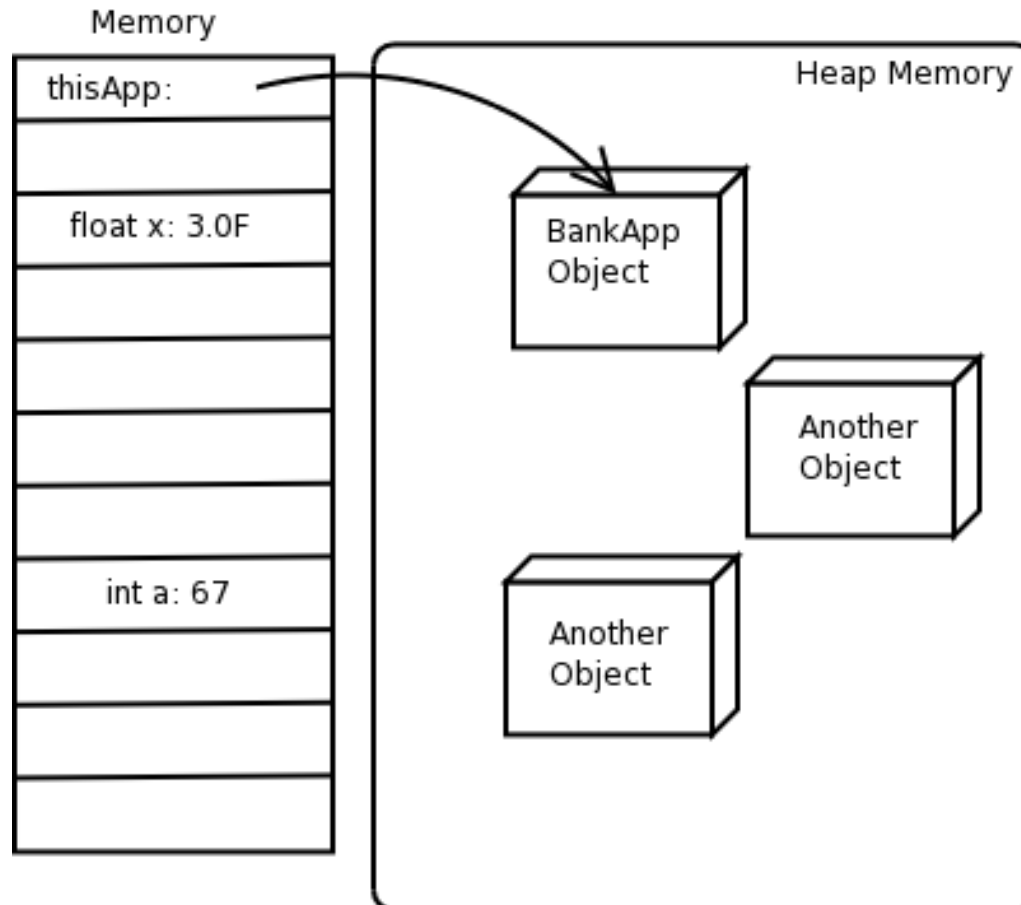
Sequence of Instantiation



A lot goes on behind the scenes when you create a new object

1. The JVM determines what type of object to create
 1. Looks at the *type* following the new operator
 2. We will look at constructors in detail a bit later
2. The JVM loads the associated class (if it is not already loaded)
3. The JVM allocates enough memory in the *heap* to hold the newly created object
4. The new object is initialized by
 1. Performing default initialization of all instance variables
 2. Executing explicit initialization of instance variables
 3. Executing the specified constructor
5. A *reference*, which we can think of as a pointer to a newly created object, is then returned and assigned to the *reference variable* `thisApp`.

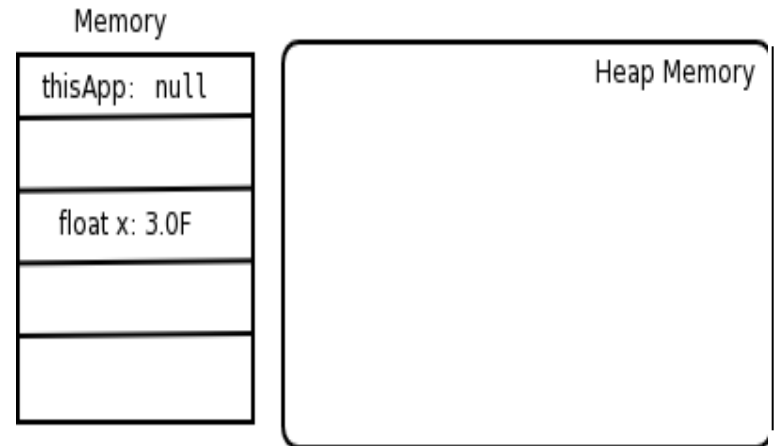
Reference Variables



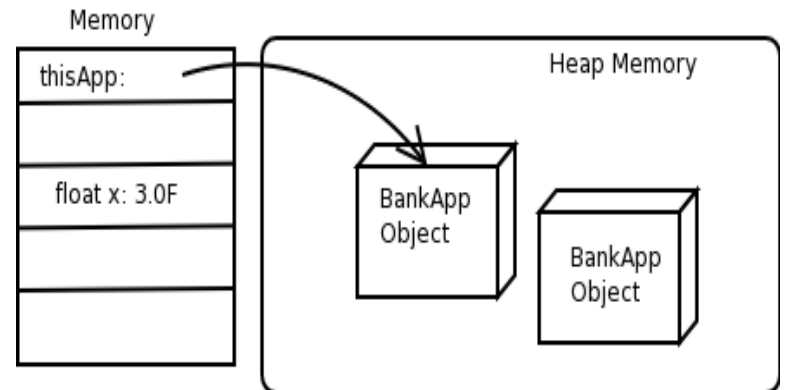
Memory allocation for reference variable thisApp

Reference Variables

```
public class BankApp {  
    public static void main(String [] args){  
        // Declare the variable.  
        BankApp thisApp = null;  
        // Create and assign the object  
        thisApp = new BankApp();  
        // Create another BankApp object  
        // don't assign IT to a variable  
        // now we have no way to refer to it!  
        new BankApp();  
    }  
}
```

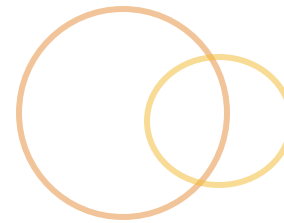


Reference variable with no associated object



Final state of the example

Object Description



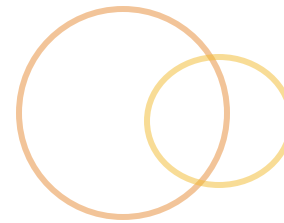
- ◎ Objects are normally described by two things:
 - ◎ Instance variables (*state*)
 - ◎ Instance methods (*behavior*)
- ◎ Both instance variables and methods are defined in class.
- ◎ Their availability for use occurs once an object has been instantiated.
- ◎ They are referred to using the *dot-notation*.

Object Description (cont.)



- ◎ Instance variables are known by many names:
 - ◎ *Attributes, States, Instance Variables, Member Variables, Members*
 - ◎ Members do not exist until an object of that type is instantiated
- ◎ Instance methods are also known by many names:
 - ◎ *Behaviors, Methods, Instance Methods*
 - ◎ Methods cannot be invoked until an object of that type is instantiated

Instance Variables



- ◉ Hold data for a specific object
 - ◉ Each object has its own memory for the instance variables
 - ◉ Instance variables exist as long as the containing object exists
 - ◉ Instance variables live and die with their instance
- ◉ Can be either primitive data types or reference types
- ◉ Instance variables are initialized several ways (in order):
 - ◉ Default initialization
 - ◉ Explicit initialization
 - ◉ Initializer
 - ◉ Constructor(s)

Instance Variables (cont.)



- ◉ The instance variable values can be adjusted either by:

- ◉ Accessing them directly

```
objectVariable.variableName = xxx;
```

or

- ◉ Invoking a method that manipulates them

```
objectVariable.setVariableName(xxx);
```

- ◉ The manner in which you access instance variables will depend on class design.

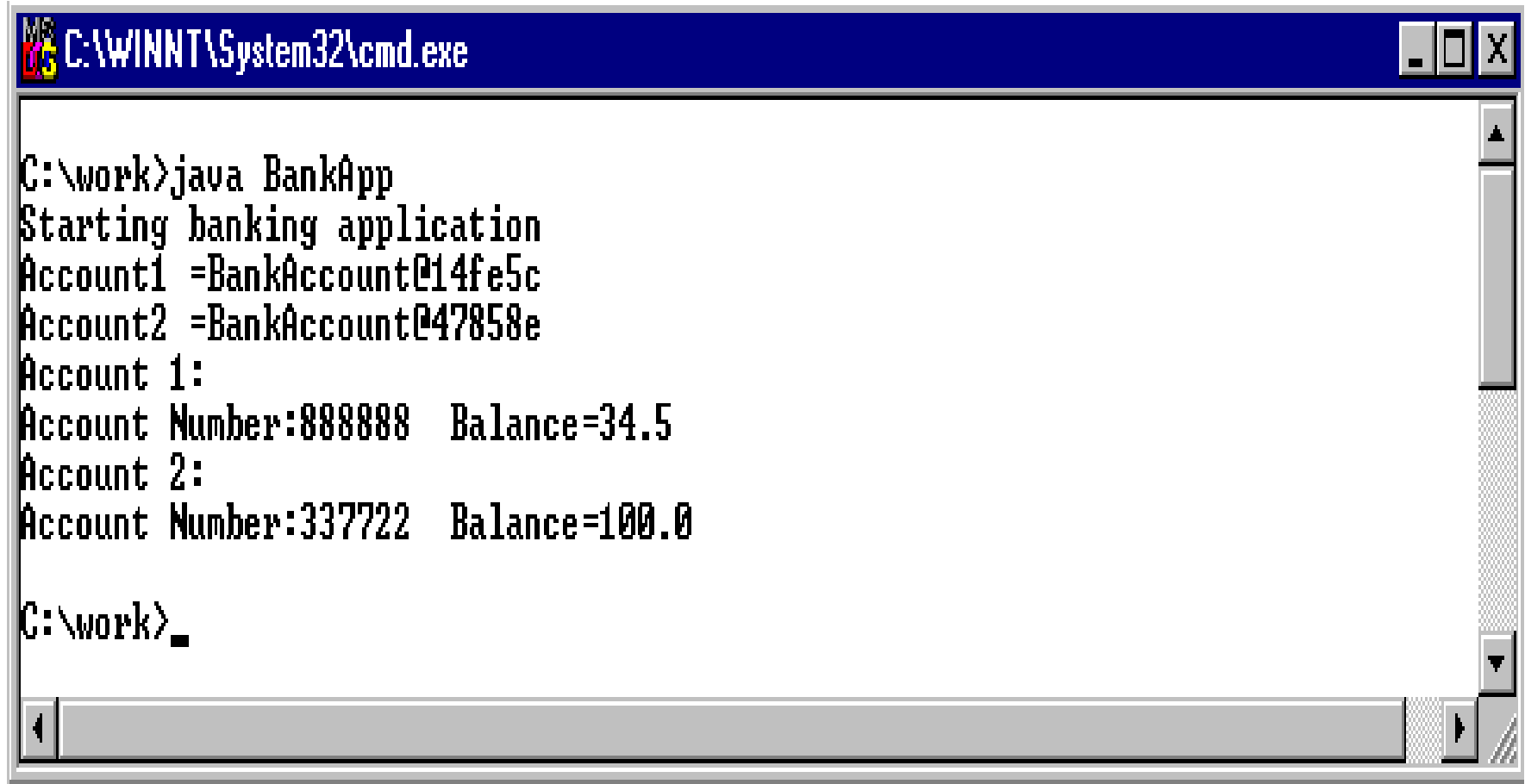
Instance Variable Example



```
class BankAccount {
    float balance;
    String acNum;
}

class BankApp {
    public static void main(String [] args) {
        BankAccount account1 = new BankAccount();
        BankAccount account2 = new BankAccount();
        System.out.println("Account1 =" + account1);
        System.out.println("Account2 =" + account2);
        // Set the balances and account numbers
        account1.balance= 34.50F;           account2.balance = 100.00F;
        account1.acNum= "888888";           account2.acNum = "337722";
        // Print out the data
        System.out.println("Account 1:\nAccount Number:" +
            account1.acNum + "    Balance=" + account1.balance);
        System.out.println("Account 2:\nAccount Number:" +
            account2.acNum + "    Balance=" + account2.balance);
    }
}
```

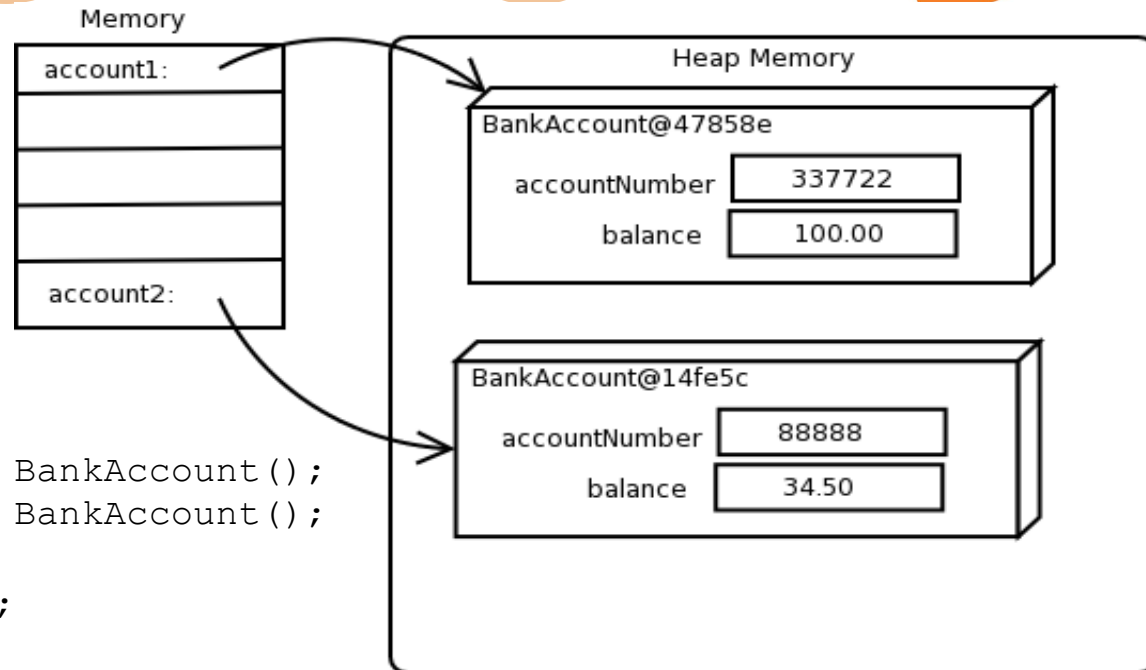
Instance Variable Example Output



```
MS-DOS C:\WINNT\System32\cmd.exe
C:\work>java BankApp
Starting banking application
Account1 =BankAccount@14fe5c
Account2 =BankAccount@47858e
Account 1:
Account Number:888888 Balance=34.5
Account 2:
Account Number:337722 Balance=100.0
C:\work>_
```

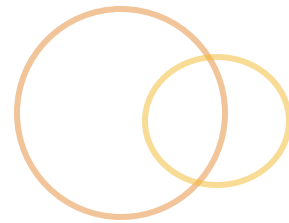
Fig. 5-4: Output from Example 5-3

Referencing Instance Variables



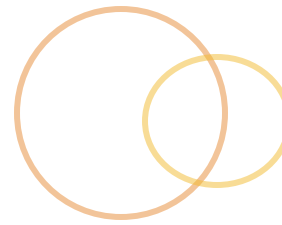
```
BankAccount account1 = new BankAccount();  
BankAccount account2 = new BankAccount();  
account1.balance= 34.50F;  
account2.balance = 100.00F;  
account1.acNum = "888888";  
account2.acNum = "337722";
```

Instance Methods



- ⦿ Perform some functionality of the object
- ⦿ Commonly associated with underlying instance variables
- ⦿ Simple instance methods might be:
 - ⦿ Accessors – retrieve values
 - ⦿ Mutators – change values
- ⦿ Instance methods are “initialized” as part of class loading
 - ⦿ Unlike instance variables, instance methods are shared by all instances of a specific class (they are “execute-only”).
 - ⦿ When an instance method is invoked, it is invoked on a specific object.
 - ⦿ Sharing the definitions is like three chefs working from one recipe book--there’s no reason to duplicate methods in memory.

Instance Methods



- ◉ The instance methods are invoked using the dot-notation
`objectVariable.setVariableName (xxx) ;`
`objectVariable.methodName (arg1, arg2 ..) ;`
- ◉ Remember, since they are associated with an instance, the instance must exist before calling a method
 - ◉ To call the method, `reference.method()`
 - ◉ Inside the method, keyword `this` refers to the current reference (what was `reference` on the outside)
- ◉ Instance methods follow the method syntax we discussed earlier

```
<access_modifier> <return> <identifier>(<parameter list>)  
float deposit(float amt)
```

- ◉ We will cover access modifiers later

Instance Method Example



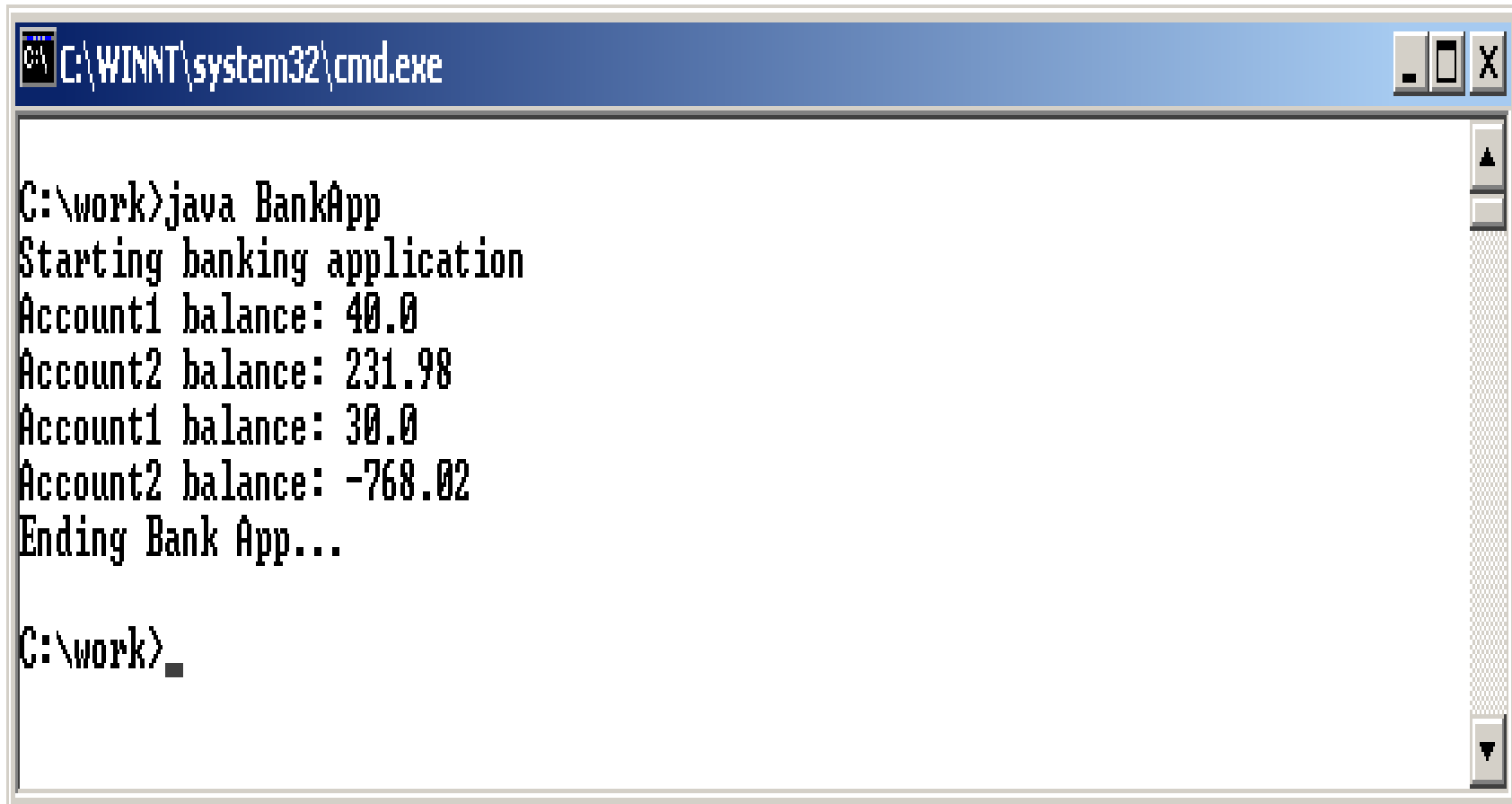
```
class BankAccount {  
    float balance;  
    String accountNumber;  
  
    float queryBalance()  
    {  
        return balance;  
    }  
  
    float deposit(float amt) {  
        balance = balance + amt;  
        return balance;  
    }  
  
    float withdraw (float amt) {  
        balance = balance - amt;  
        return balance;  
    }  
}
```

Instance Method Example (cont.)



```
class BankApp {
    public static void main(String [] args) {
        System.out.println("Starting banking application");
        // create two new bank accounts.
        BankAccount account1 = new BankAccount();
        BankAccount account2 = new BankAccount();
        // Make deposits into each
        account1.deposit(40.00F);
        account2.deposit(231.98F);
        // Display their balances
        System.out.println("Account1 balance: " + account1.queryBalance());
        System.out.println("Account2 balance: " + account2.queryBalance());
        // Make a withdrawal from each account
        account1.withdraw(10.00F);
        account2.withdraw(1000.00F);
        // Display their balances
        System.out.println("Account1 balance: " + account1.queryBalance());
        System.out.println("Account2 balance: " + account2.queryBalance());
        System.out.println("Ending Bank App...");
    }
}
```


Instance Method Example Output



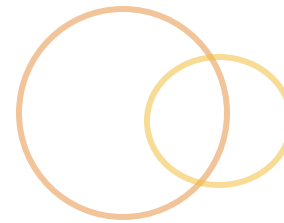
```
C:\WINNT\system32\cmd.exe

C:\work>java BankApp
Starting banking application
Account1 balance: 40.0
Account2 balance: 231.98
Account1 balance: 30.0
Account2 balance: -768.02
Ending Bank App...

C:\work>
```

Output from example

Business Rules



- Policies, procedures, and workflows automated by the application are commonly referred to as business rules.
 - It is important to ensure objects in the application conform to those rules.
 - Business rules are represented in program logic.
- Business rules can also place execution and environment constraints on an application.
- It should not be the programmer's job to determine business rules.
 - Domain experts are responsible for defining the rules.

Business Rules

- ◎ Methods implement business rules.
- ◎ Some business rules are called guards:
 - ◎ They guard against a method executing improperly
 - ◎ OOP suggests incorporation of:
 - ◎ Preconditions: boolean conditions that must be true before the method can be executed
 - ◎ Postconditions: boolean conditions that must be true after a method executes
 - ◎ Invariant conditions: boolean conditions that must always be true

Business Rules and Methods



```
class BankAccount {
    float balance;
    String accountNumber;
    int accountStatus;

    float deposit(float amt) {
        if (accountStatus != 0) {
            return 0.0F;
        }
        balance = balance + amt;
        return balance;
    }

    float withdraw (float amt) {
        if (accountStatus != 0) {
            return 0.0F;
        }
        if (amt <= balance) {
            balance = balance - amt;
        }
        return balance;
    }
}
```

Initialization of Instance Variables

- ⦿ All variables must be initialized before they can be read.
 - ⦿ This rule is strictly and obviously enforced with local variables.
 - ⦿ This rule is not obvious with instance variables, because instance variables are always initialized.
 - ⦿ Implicit initialization sets value to “zero”
 - ⦿ You can provide more explicit initialization

Initialization of Instance Variables



- ◉ Default initialization
 - ◉ Unavoidable/automatic
- ◉ Initialize in the declaration
 - ◉ `int x = 99;`
 - ◉ Referred to as “explicit initialization”
- ◉ Initialize a variable in an initializer
 - ◉ Code in an unnamed, unlabeled block
- ◉ Initialize a variable in the constructor
 - ◉ Allows arguments to constructor to be used to calculate initializing value

Initialization of Instance Variables (cont.)



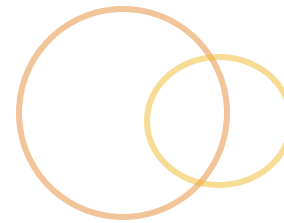
- Initialization mechanisms have different results
 - Default initialization
 - Reference variables, including `String` variables, are all initialized to `null`
 - Numeric values are initialized to the appropriate zero value and
 - `boolean` types are initialized to `false`
 - Explicit initialization
 - Variables initialized to some specific value
 - Value may be computed, but limited data are available for this computation
 - Default initialization is overwritten
 - Initializer/Constructor initialization
 - Initializer follows explicit, constructor follows initializer
 - Can overwrite previous initializations
 - Values may be computed, constructor args may be used
- Normally, instance variables are initialized in the constructor

Explicit Initialization Example



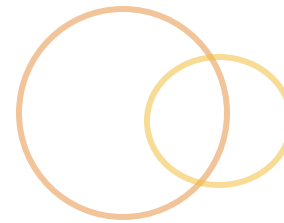
```
class BankAccount {  
    float balance = -1.0F;  
    String accountNumber = "NotSet";  
    int accountStatus = -1;  
    char accountType = " "  
}
```


Constructors



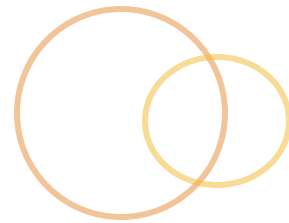
- Objects are created through a `new SomeType()` call
- After the memory has been created, the object is initialized in the constructor
- Constructors may be thought of as initialization methods
 - Note there is no return value

Constructor Purpose



- ◎ The purpose of the constructor is to initialize the newly created object
 - ◎ Allows correct instance variable initialization
 - ◎ Any other initialization or startup code can be executed
 - ◎ Perform complex initialization logic that cannot be done as an explicit initialization, e.g., loops
- ◎ Constructors allow us to call `new`
 - ◎ What follows the keyword “new” must match the signature of a constructor
- ◎ If you provide no constructors, the compiler provides one
 - ◎ Referred to as the default constructor
 - ◎ No arguments
 - ◎ Compiler doesn't know about your class' semantics, so there is no behavior in the default constructor

Constructor Rules



Constructors must abide by some specific rules

- ⦿ The constructor *always*
 - ⦿ Has the same name as the class
 - ⦿ Remember Java is case sensitive
- ⦿ Constructors can be overloaded
 - ⦿ Similar to method overloading
 - ⦿ May be multiple constructors with different argument lists
- ⦿ Does not declare a return value
 - ⦿ This is not the same as returning `void`
 - ⦿ A constructor initializes the newly created object

Constructor Example

```
class BankAccount {
    float balance = -1.0F;
    String accountNumber = "NotSet";
    int accountStatus = -1;
    char accountType = " ";

    BankAccount(String num, char type) {
        accountNumber = num;
        accountType = type;
        balance = 0.0F;
        accountStatus = (type == 'p')? 100: 0;
    }
    BankAccount(String num, char type, float bal) {
        accountNumber = num;
        accountType = type;
        balance = bal;
        accountStatus = (type == 'p')? 100: 0;
    }
    BankAccount(String num) {
        accountNumber = num;
    }
    /* -- rest of class -- */
}
```

Proper Constructor Form



- Typically a class defines multiple constructors.
 - Each constructor varies by argument list.
 - Though the constructors are different, they should perform the same level of initialization.
- Having many constructors
 - Provides flexibility
 - Can be error prone if done wrong
- Constructors can refer to other constructors
 - To minimize redundant code
 - Provide centralized initialization
 - Simplify maintenance

Proper Constructor Form (cont.)



- ◉ When referring to other constructors
 - ◉ Utilize a built-in mechanism - `this (...)`
 - ◉ Think of `this (...)` as a constructor calling another constructor
 - ◉ Like a method call
 - ◉ JVM determines which constructor to call
- ◉ Use `this (...)` as the first execution in your constructor
- ◉ Can perform other operations once `this (...)` “returns”

Proper Constructor Form



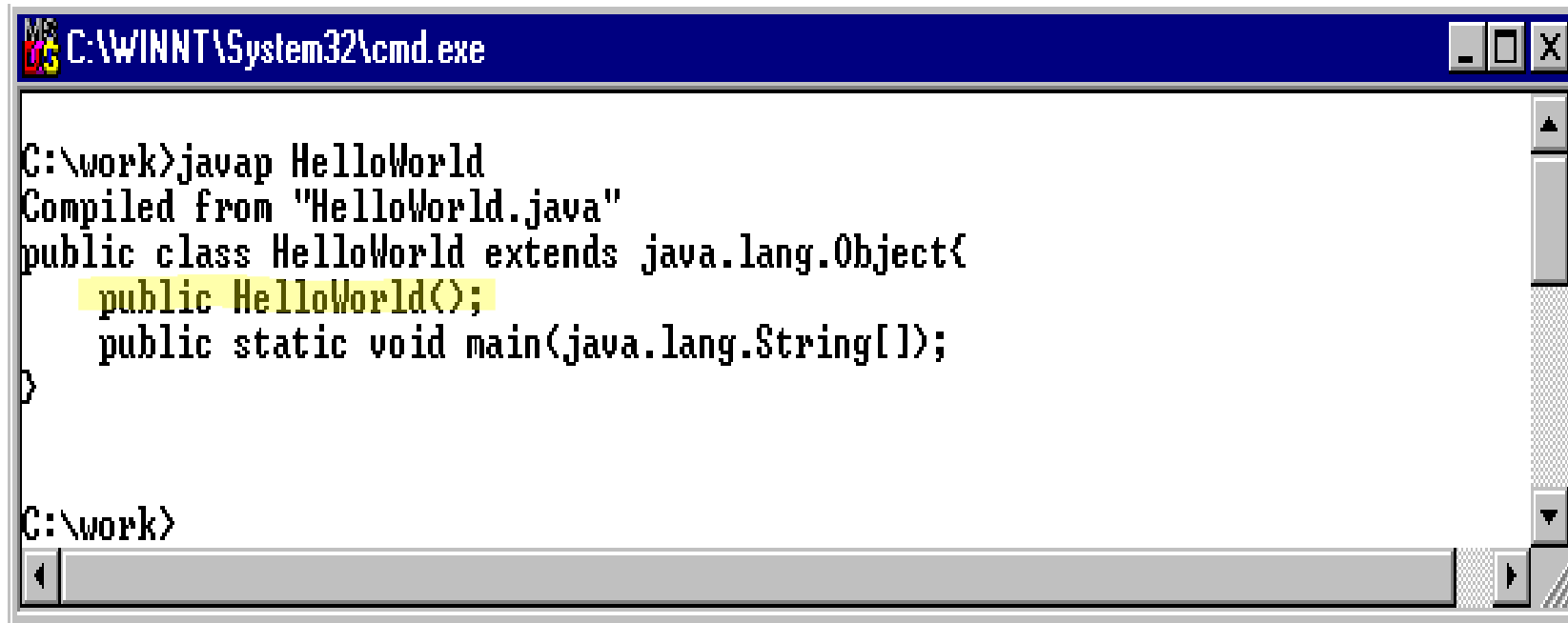
```
class BankAccount {
    float balance = -1.0F;
    String accountNumber = "NotSet";
    int accountStatus = -1;
    char accountType = " ";

    BankAccount(String num, char type, float bal) {
        accountNumber = num;
        accountType = type;
        balance = bal;
        accountStatus = (type == 'p')? 100: 0;
    }
    BankAccount(String num, char type) {
        this(num, type, 0.0F);
    }
    BankAccount(String num) {
        this (num, 'p');
    }
    /* -- rest of class -- */
}
```

The Default Constructor



In the first module, we used the disassembler (`javap`) to look into our `HelloWorld` class



```
MS-DOS Batch File C:\WINNT\System32\cmd.exe
C:\work>javap HelloWorld
Compiled from "HelloWorld.java"
public class HelloWorld extends java.lang.Object{
    public HelloWorld();
    public static void main(java.lang.String[]);
}
```


The Default Constructor



```
//This compiles and runs
class Test1 {
    int x;
    public static void main(String [] args) {
        Test1 t = new Test1(); // this is the default constructor
        System.out.println(t);
    }
}

//This does not compile
class Test2 {
    int x;
    // Adding this constructor prevents the default
    // constructor is not provided
    Test2(int xs) {
        x = xs;
    }
    public static void main(String [] args) {
        Test2 t = new Test2(); // this constructor no longer exists.
        System.out.println(t);
    }
}
```

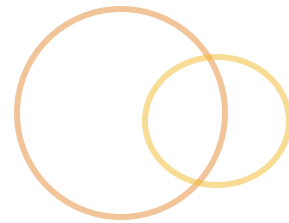
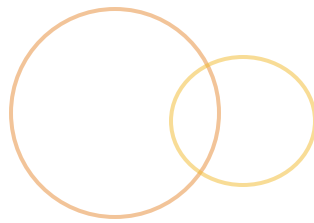
The Instance_INITIALIZER



- ◎ Initializers are invoked prior to constructors.
- ◎ Multiple initializers are permitted and are executed from top to bottom of the class.
- ◎ The syntax is simply an unadorned block in the class

```
class ThingOne {  
    int someNumber;  
    {  
        someNumber = (int)(Math.random() * 1000);  
        if (Math.random() > 0.9) { someNumber = 0; }  
    }  
    /* Rest of class definition */  
}
```

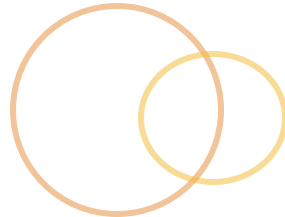
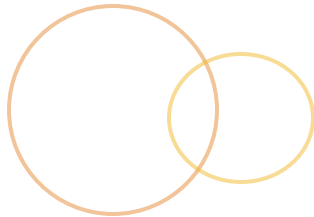
Summary



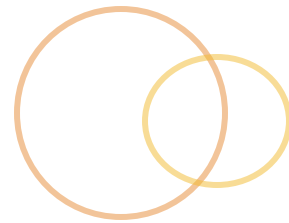
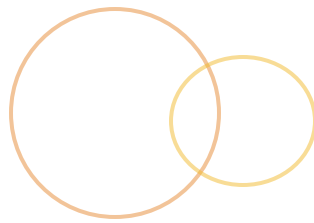
We covered:

- ⦿ Using the `new` operator to create objects
- ⦿ Describing how reference variables work
- ⦿ Using instance variables and methods
- ⦿ Describing and using constructors

Objects And Classes (Part 2)



Objectives



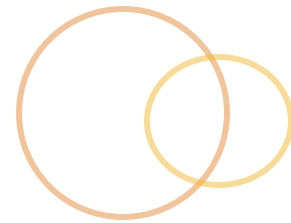
At the end of this section, you should be able to use:

- Use the **new** operator to create object
- Describe how reference variables work
- Use instance variables and methods
- Define Business Rules
- Use the **final** keyword with variables
- Use wrapper classes

Creating Objects

- ⦿ Every OOP language must have a way of creating new objects from the class definition on demand
- ⦿ Java uses the instantiation mechanism that it inherited from C++ -- the **new** operator
- ⦿ There is only one way to create an object in Java – by using the **new** operator

The new Operator



```
public class BankApp {  
    public static void main(String [] args)  
    {  
        // create the BankApp object.  
        BankApp thisApp = new BankApp() ;  
    }  
}
```

The JVM Builds Objects In Memory



- ⦿ The JVM class loader finds the appropriate class definition and loads it
- ⦿ Heap memory is allocated to hold the newly created object
- ⦿ The new object is initialized by executing the code in a special function called a constructor, which we will discuss later.
- ⦿ A *reference*, which we can think of as a pointer to a newly created object, is then returned and assigned to the *reference variable* **thisApp**.

Object Reference Semantics



- Objects are accessed using references
- References are variables containing a “pointer” to an object
 - A number, often a 32-bit integer, identifying the heap location of your object
 - The reference value is hidden from you
- Copying a reference value *only* copies the reference value
 - You do not actually copy the underlying object
 - The end result is two references with the same value, referring to the same object in the heap

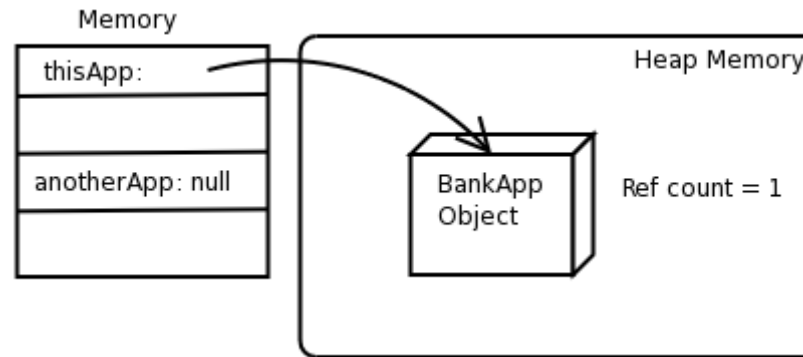
Object Reference Semantics Example



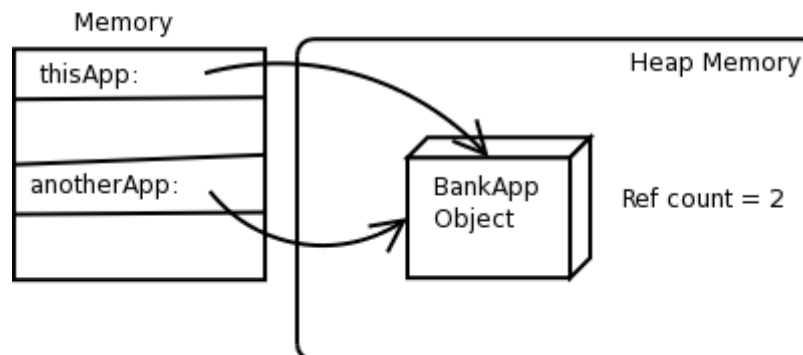
Example: Reference variable assignment

```
class BankApp {  
    public static void main(String [] args) {  
        // Declare the reference variables  
        BankApp thisApp = null;  
        Bankapp anotherApp = null;  
        // Create the object and assign the reference  
value  
        thisApp = new BankApp() ;  
        // Now assignment of reference variables  
        anotherApp = thisApp;  
    }  
}
```

Object Reference Semantics

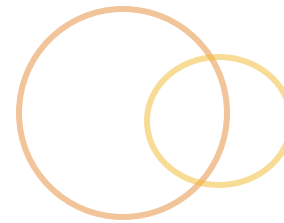


```
thisApp = new BankApp();  
anotherApp = null;
```



```
anotherApp = thisApp
```

Instance Variables



- ◉ Instance variables make sense to us at an intuitive level
- ◉ We know that every bank account has a property called *balance* and each bank account has its own balance
- ◉ We can think of instance variables as the data that describes a particular object.

Instance Variable Properties



- ◎ Instance variables hold data *for a specific object*.
- ◎ Instance variables are defined in the class definition.
- ◎ Instance variables do not exist until an object of that type is instantiated.
- ◎ Each object has its own copy of the instance variables.
- ◎ Instance variables can be either primitive data types or reference types.

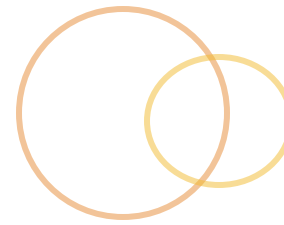
Referencing Instance Variables



- Since instance variables exist inside objects, we have to refer to the object containing instance variable we want to access
- The syntax for referring to an instance variable is:

```
Object objectVariable = new  
Object();  
objectVariable.variableName
```

Instance Methods



- Objects have methods associated with them as well as variables.

```
class BankAccount {  
    float balance;  
    float queryBalance() {  
        return balance;  
    }  
  
    float deposit(float amt) {  
        balance = balance + amt;  
        return balance;  
    }  
}
```

Invoking instance methods



- Invoking a method on an object can be thought of as *asking the object to run its local copy of the method on the processor available to it*
- From a more theoretical OO perspective, when we invoke a method on an object, we are sending a message to that object to get the object to exhibit some specific behavior

Invoking instance methods



- We invoke a method on an object by using a similar syntax to referencing an instance variable:

```
objectVariable.methodName (arg1, arg2..)
```

Business rules



- *Application correctness* means that the proper business rules for that problem domain are implemented in the code
- For example, if our **BankAccount** objects in code represent real bank accounts, we should not be able to do things in our code that we are not allowed to do in the real world
- Since these rules are about the business that our application is automating, they are called *business rules*

Object Life Cycle & Garbage Collection



- ◎ Java provides built-in memory management
 - ◎ Used for allocating memory
 - ◎ Used for deallocating memory – a.k.a. garbage collection
- ◎ The garbage collector (*gc*) is a daemon (background) thread that runs in the virtual machine
- ◎ There are many different types of garbage collection algorithms; in general the *gc* checks on a regular basis which objects have become moribund

Object Life Cycle & Garbage Collection (cont.)



- ◉ The garbage collector frees up occupied but unreferenced memory automatically
 - ◉ An object is considered to be referenced as long as there is at least one usable reference to it
 - ◉ As soon as zero accessible references exist, then the object can be garbage collected
- ◉ Objects can lose accessible references when:
 - ◉ Their reference variables become `null` (through assignment)
 - ◉ Their reference variables are reassigned with a new value
 - ◉ Their reference variables go out of scope (local reference variables)

Object Life Cycle & Garbage Collection Example

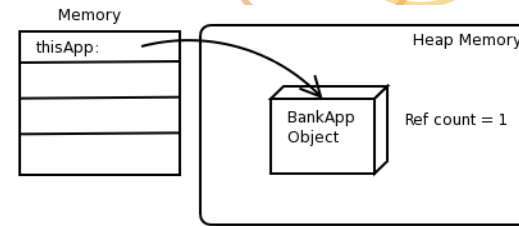


```
class BankApp {
    public static void main(String [] args) {
        // thisApp is now a local variable to this block
        {
            BankApp thisApp = new BankApp();
            // BankApp object now has a reference count of 1
            {
                BankApp anotherApp = thisApp;
                // BankApp object now has a reference count of 2
                // Step two in fig. 5-9
            }
            // anotherApp is out of scope. Reference count is 1 again
            // Step three in fig. 5-9
        }
        // thisApp is now out of scope, Reference count is 0
        // BankApp object is moribund waiting for garbage collection
        // Step four in fig. 5-9
    } // end of main method
}
```

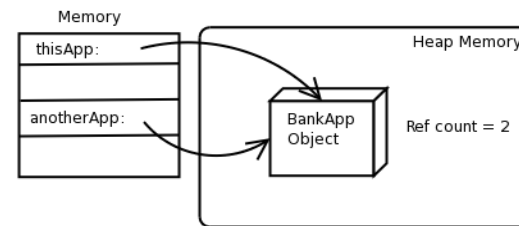
Object Life Cycle & Garbage Collection



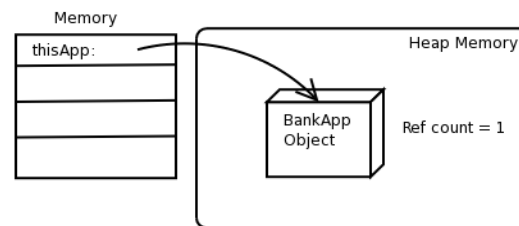
```
void f() {  
    BankApp thisApp  
        = new  
BankApp();  
    { // new scope  
        BankApp  
anotherApp  
        = thisApp;  
    }  
}
```



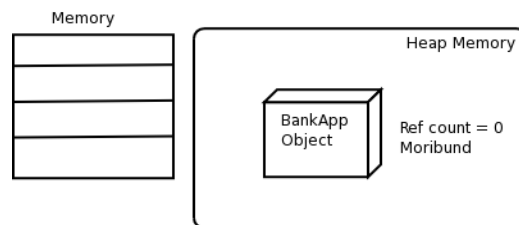
Step One



Step Two

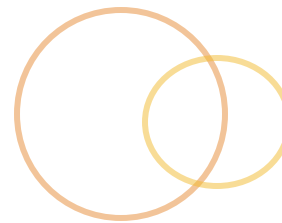


Step Three



Step Four

The `finalize()` Method



- Before an object is garbage collected, the JVM system calls its `finalize()` method.
- The `finalize()` method gives the object a chance to return allocated or in-use system resources
 - Kind of the opposite of a constructor
 - There is no guarantee that the `finalize()` method will actually be called
 - The original intent of the `finalize()` method was to free up resources allocated by native code, e.g., file handles, windows
- You can include a `finalize` method in your class

```
protected void finalize()
```
- Finalize is generally not recommended*

finalize() Method Example



```
class BankApp {
    BankApp() {
        System.out.println("Creating
BankApp");
    }
    protected void finalize() throws
Throwable {
        System.out.println("Finalizing
BankApp");
        return;
    }
    public static void main(String [] args)
{
        new BankApp();
    }
}
```


Comparing Reference Variables



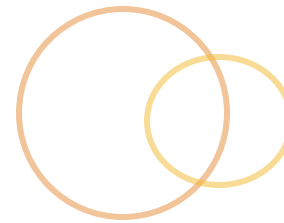
- ◎ Reference variables contain a value which describes the location of an object in the heap.
- ◎ This value is hidden from you.
- ◎ If you want to compare the value of two references, you can use the equality operator
 - ◎ Use the standard `==` operator for this comparison
 - ◎ `if (refVar1 == refVar2)`
 - ◎ `if (refVar1 != refVar2)`

Comparing Reference Variables Example



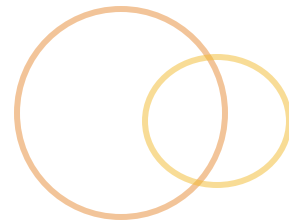
```
class Test {  
    public static void main(String [] args) {  
        String s1 = new String("Hello");  
        String s2 = new String("Hello");  
        System.out.println  
            ("Before assignment: s1 == s2 ->" + (s1 == s2));  
        String s3 = s1;  
        System.out.println  
            ("After assignment: s1 == s3 ->" + (s1 == s3));  
    }  
}
```

Representing Text



- ◎ Java has classes for dealing with strings of text
 - ◎ `java.lang.String`
 - ◎ `java.lang.StringBuilder` & `java.lang.StringBuffer`
- ◎ String objects can be
 - ◎ Literals
 - `String literal = "String Literal";`
 - ◎ Objects
 - `String object = new String("String Object");`
- ◎ The String class provides many methods
 - ◎ `length()`
 - ◎ `substring()`
 - ◎ `toLowerCase()` / `toUpperCase()`
 - ◎ `startsWith(String)`
 - ◎ Etc.

Wrapper Classes



- ◉ Sometimes it is useful to treat primitives as objects.
- ◉ Java provides wrapper classes for all primitive types:
 - ◉ `java.lang.Long`
 - ◉ `java.lang.Integer`
 - ◉ `java.lang.Short`
 - ◉ `java.lang.Boolean`
 - ◉ etc.
- ◉ The wrapper classes provide some useful functionality like:
 - ◉ Converting primitives to and from `String`
 - ◉ Retrieving `System` properties as the primitive value

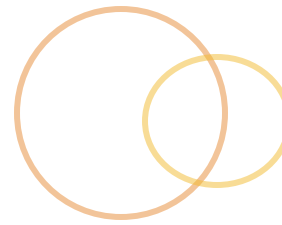
Wrapper Classes Example



Wrapper classes

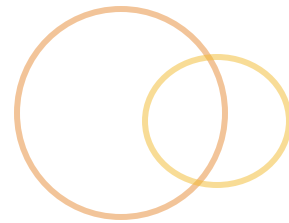
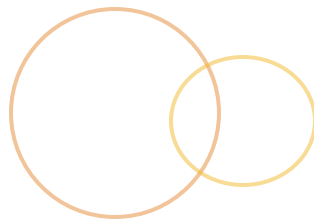
```
class Test {  
    public static void main(String [] args) {  
        String s = "7839276";  
        long var = Long.parseLong(s);  
        System.out.println("Value of var is "+var);  
        // Create a Long object to wrap this value  
        Long obj = new Long(var);  
        // This is an object but we can still get the data  
        System.out.println("obj wraps "+ obj.longValue());  
    }  
}
```

Autoboxing/unboxing



- Since Java 1.5, assignment between primitives and their wrappers has been transparent
 - `// executes as x = new Integer(5);`
`Integer x = 5;`
 - `// executes as y = new`
`Integer(5).intValue();`
`int y = new Integer(5);`
- Take care, the computations still happen, which can be wasteful.
- These features are particularly valuable with the collections API
 - Allows us to store primitives in containers made for objects

Summary



We covered using:

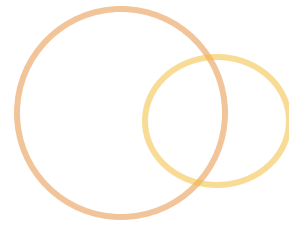
- ⦿ The **new** operator
- ⦿ Reference variables work
- ⦿ Instance variables and methods
- ⦿ Business Rules Definition
- ⦿ Use the **final** keyword with variables
- ⦿ Use wrapper classes

Finding and Using Common Classes

Navigating the Application Programming Interface

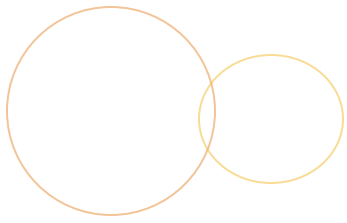


Presentation Topics



In this presentation, we will discuss:

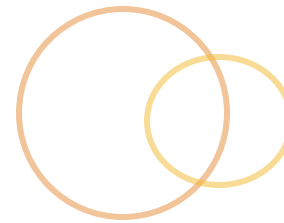
- 🕒 Navigating the Java API
- 🕒 Comments
- 🕒 Naming Conventions
- 🕒 Using String and StringBuilder objects
- 🕒 Using Arrays



There are 3 APIs (Application Programming Interfaces) associated with the Java Programming Language

- ◎ Java Standard Edition
- ◎ Java Enterprise Edition
- ◎ Java Micro Edition

Java SE 7.0 API



The Java API lists all of the Packages, Classes, Methods, and Variables that ship with the Java SE 7.0 JDK and or JRE

Java also includes a nifty tool called *javadoc* that will create the same API for custom classes and provide mechanisms to comment your programs in a JavaDoc friendly way.

Learning to navigate the API is the a major step to understanding the language.

Oracle Publishes Each New Version of Java With A New API



- ⦿ <http://docs.oracle.com/javase/7/docs/api/>
Introduced in 1.1
- ⦿ Lists overview of each package
- ⦿ Lists all deprecated methods
 - ⦿ Deprecated methods are methods that are no longer supported
 - ⦿ Deprecated methods are not removed so that new version won't break older code
 - ⦿ Deprecated methods usually list the new version of the method to use if there is one:

Deprecated Method Example



setDate

```
@Deprecated  
public void setDate(int date)
```

Deprecated. As of JDK version 1.1, replaced by `Calendar.set(Calendar.DAY_OF_MONTH, int date)`.

Sets the day of the month of this `Date` object to the specified value. This `Date` object is modified so that it represents a point in time within the specified day of the month, with the year, month, hour, minute, and second the same as before, as interpreted in the local time zone. If the date was April 30, for example, and the date is set to 31, then it will be treated as if it were on May 1, because April has only 30 days.

Parameters:

`date` - the day of the month value between 1-31.

See Also:

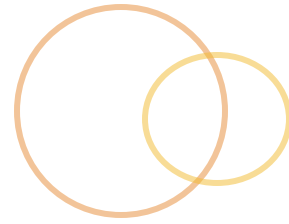
`Calendar`

API is Organized Using Frames



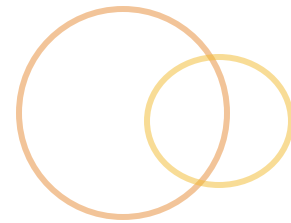
- The JAVA API is organized by Packages, Classes, and Member Methods and Variables.
- Top of main frame also lists each member by name in an index.
- API provides hierarchy tree for each class.

Easily Navigated



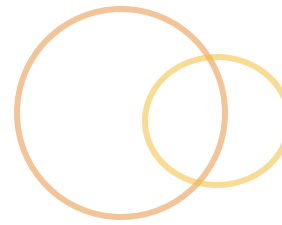
- Use the top left pane to locate the package or sub-package you are looking for.
- Use the bottom left pane to locate the class you are looking for.
- Use the main pane to examine the class and find methods and arguments and/or variables to use in your program.

Java Comments



- Single Line Comments
- Multiple Line Comments
- javadoc Comments

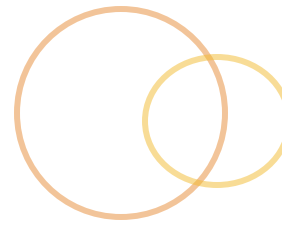
Single Line Comment



- Java uses C style comments for single lines
- When the Java compiler encounters a `//`, it treats everything that follows the `//` up to the end of the line as a comment.

```
//This is an example of a single line comment  
//It only spans one line  
//Each line must be commented individually  
Int I = 10; //This is a comment after a statement
```

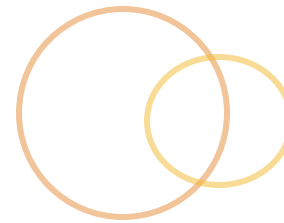
Multi-line Comments



- When the compiler encounters the pair of characters `/*` (which together are called the *opening delimiter* of the comment), it treats everything it encounters up to the sequence of characters `*/` (the *closing delimiter*) as a comment

```
/*   This is the start of a Multi-line comment
*       This is the second line of a Multi-line
comment
*/
```

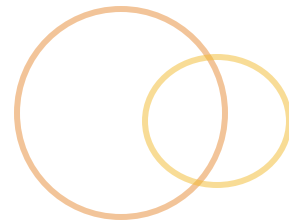
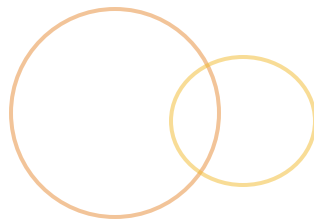
javadoc Comments



- javadoc comments have the same syntax as multi-line comments
- The compiler creates an HTML document from javadoc comments that mirror the Java API format

```
/**  
 * First line describing SomeClass that is  
 * used in the index.  
 * <p>  
 * HTML <strong>formatted</strong> comments  
 *  
 */
```

No Bloat



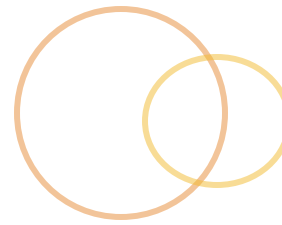
- ◎ The javac compiler strips comments out of the source code when it compiles the “.java” file to the “.class” file
- ◎ Comments can never bloat our application so you should use them liberally to document your code
- ◎ It may seem obvious what your code does when you write it. It might not be obvious when you go to make changes six months later

The SDK includes javadoc



- **javadoc** is designed so that a programmer can document classes, methods, fields and packages in an application, and then leave it up to **javadoc** to format, organize, and cross-reference the resulting documentation.
- **javadoc** is an automated documentation generation tool.

Naming Conventions



⦿ Packages (lowercase)

- ⦿ The standard way to name packages is the reverse dot notation of the organization:

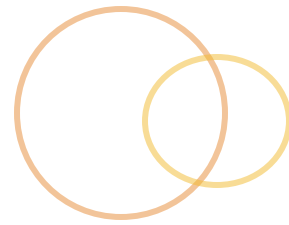
`com.auto_owners.packageName.subPackageName`

- ⦿ Hyphens are not allowed so they are replaced with an underscore

First letter of each noun should be capitalized

- ⦿ Classes are represented as nouns, never verbs
- ⦿ Interfaces are adjectives and should describe the enforced behavior of the noun/class
- ⦿ Enums should follow Class and Constant conventions. Capitalized first letter nouns.

Naming Conventions



Methods

- Start with lowercase letter and capitalize first letter in subsequent words
- If not an accessor method, name should be a verb describing what method does, e.g., `free()`, `list()`, `freeMemory()`
- Accessor method should always start with `get` and `set` followed by the variable they are reading or writing: e.g., `getColor()` `setColor`

Attributes and local variable names

- Start with lowercase letter and capitalize first letter in subsequent words
- Always use full names, no abbreviations: e.g., `color`, `tires`

String vs. StringBuilder vs. StringBuffer



- Strings are arguably one of the most important Classes in Java
- Strings are immutable and final. Once created a String never leave the Classloader
- If Strings were not Immutable, any program could change the name of a String that represented a file name or loaded class, possibly gaining access to unprivileged objects.
- String overrides the “+” operator allowing Strings to be added together
 - `String fullName = “Peter ”+”Andrew ”+”Campbell ”+40;`
- StringBuilder and StringBuffer eliminate excess String garbage in the heap.
- StringBuilder and StringBuffer can be modified without creating new Objects in the heap.

Overloading the “+” Operator



- ⦿ The “+” operator is not really overloaded, it just acts like it.
- ⦿ “+” is a concatenation operator and an addition operator depending on what it is acting upon.
- ⦿ Every Class has a toString() method which is called when the “+” is used to convert to a String
- ⦿ Primitives use their Object counterpart to get their toString() value:

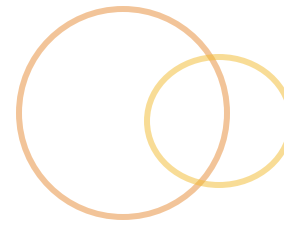
```
int i = 5;  
String five = "five == "+ i;  
String five = "five ==  
"+Integer.valueOf(5).toString();
```

StringBuffer vs. StringBuilder



- StringBuffer was the de facto way to manipulate Strings and reserve memory before Java 1.5
- StringBuffer is synchronized, meaning that when you create a StringBuffer it cannot be shared efficiently within a program.
- StringBuilder is essentially identical to StringBuffer except it is not synchronized for reads and is therefore much more efficient.
- StringBuilder should be used over StringBuffer whenever possible.

Creation Examples



```
String s = "Always Around";  
String alsoAround = s.substring(8,13);  
String forever = s + " Forever\n";  
StringBuffer sBuffer = new  
StringBuffer("CollectMe");  
// Still creates "CollectMe" as a String in  
ClassLoader but subsequent manipulations don't add  
to ClassLoader  
sBuffer.reverse();  
StringBuilder sBuilder = new  
StringBuilder("CollectMe");  
// Still creates "CollectMe" as a String in  
ClassLoader but subsequent manipulations don't add  
to ClassLoader  
sBuilder.reverse();
```

Java Offers Two Types of Object Containers

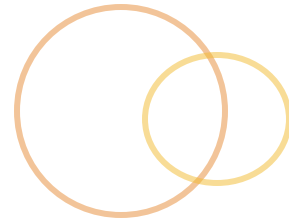
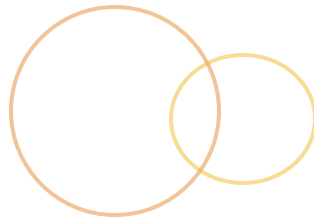


- Arrays and Collections

- Arrays are simple storage vessels with a fixed size and costly growth.

- Collections offer numerous possibility for flexible containment of Objects.

- Collections allow for resizing and are less expensive when their size changes.
 - Programs can choose the best collection for the job.

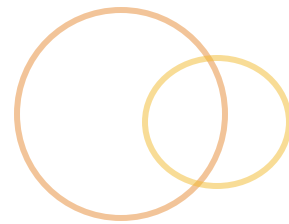


- Arrays have been replaced by the Collections Framework
- Still necessary at times but inflexible and costly

```
int[] intArray = new int[5];  
Int[] intArray = {1,2,3,4,5};  
Int[] intArray = new int[]{1,2,3,4,5};
```

```
String[] stringArray = new String[5];  
String[] stringArray = {"a","b","c","d","e"};  
String[] stringArray = new String[]  
    {"a","b","c","d","e"};
```

Arrays as Objects



- ◉ Earlier we discussed arrays as
 - ◉ Basic “data structure”
 - ◉ With an inherent attribute called `length`
- ◉ Arrays in Java are objects
- ◉ Objects are created with the `new` keyword
 - ◉ Arrays have a literal form too
- ◉ Objects contain attributes; arrays contain attributes called elements
- ◉ There is no “constructor” when dealing with an array

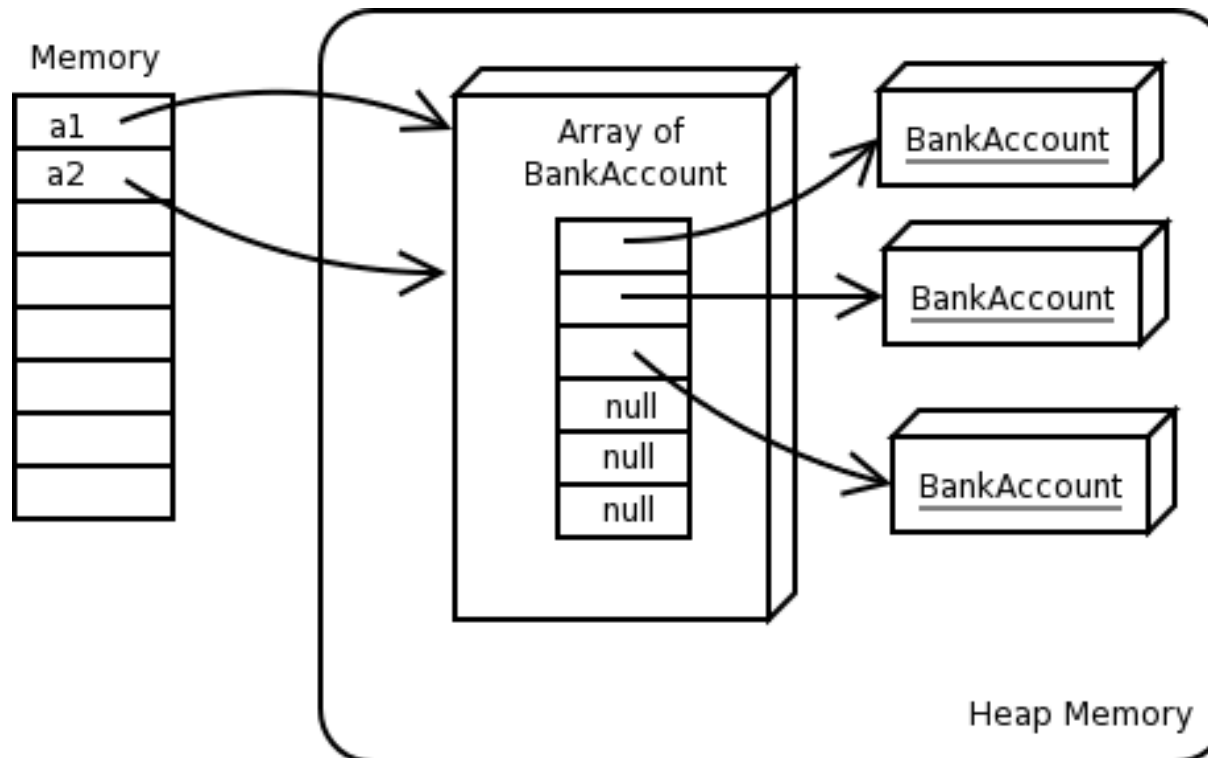
Arrays as Objects Example



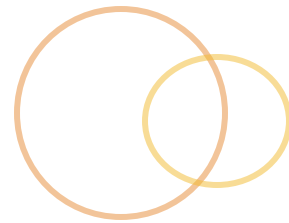
Creating arrays

```
class Test {  
    public static void main(String [] args) {  
        // create the array references  
        BankAccount [] a1;  
        BankAccount a2 [];  
        // create an array  
        a1 = new BankAccount[6];  
        for (int k = 0; k < 3; k++) {  
            a1[k] = new BankAccount();  
        }  
        // make a1 and a2 point to the same array  
        a2 = a1;  
    }  
}
```

Arrays as Objects



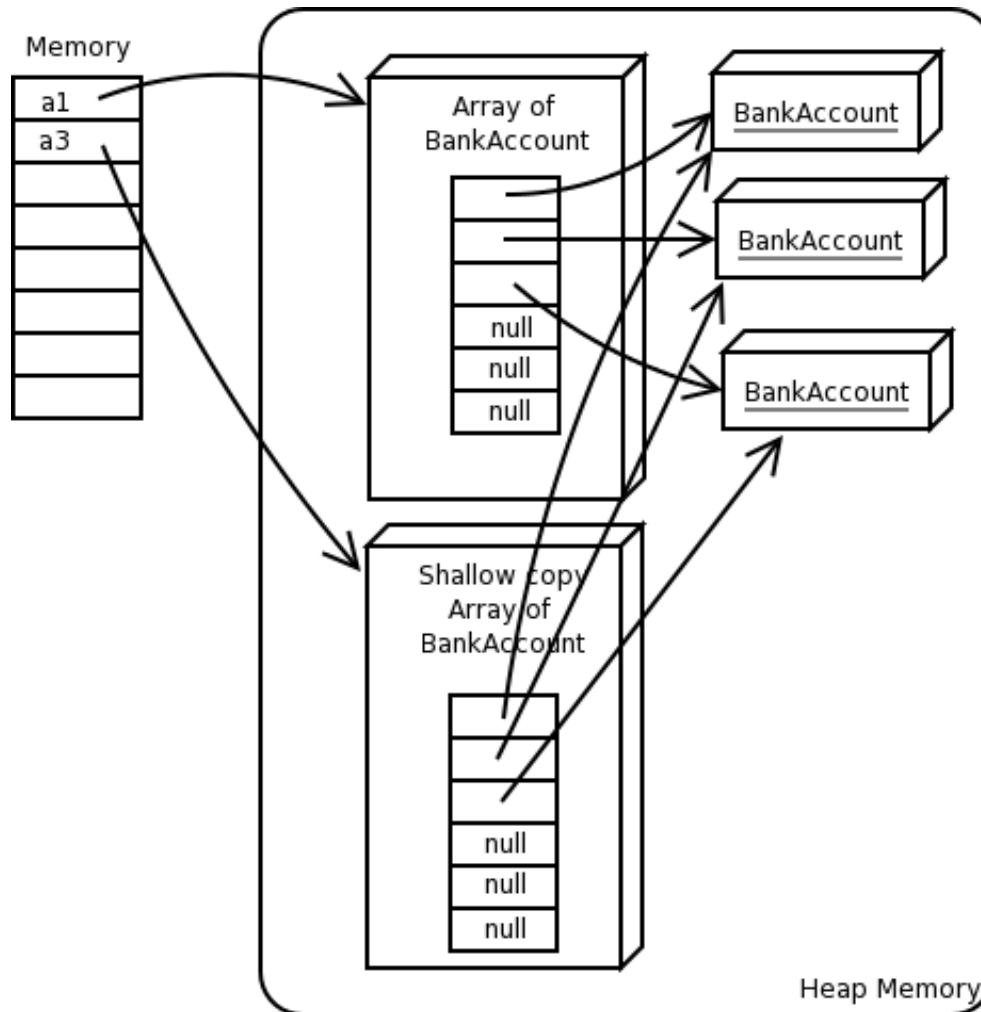
Copying an Array



Copying arrays

```
class Test {  
    public static void main(String [] args) {  
        // create the array references  
        BankAccount [] a1;  
        BankAccount a2[];  
        BankAccount [] a3;  
        // create an array  
        a1 = new BankAccount[6];  
        for (int k = 0; k < 3; k++) {  
            a1[k] = new BankAccount();  
        }  
        // make a2 into a copy of a1 - manual/wrong way  
        a2 = new BankAccount[a1.length];  
        for (int k = 0; k < a1.length; k++) {  
            a2[k] = a1[k];  
        }  
        a3 = new BankAccount[a1.length];  
        // make a3 into a copy of a1 -- easy way  
        System.arraycopy(a1,0,a3,0,a1.length) ;  
    }  
}
```

Copying an Array



Result of shallow copy

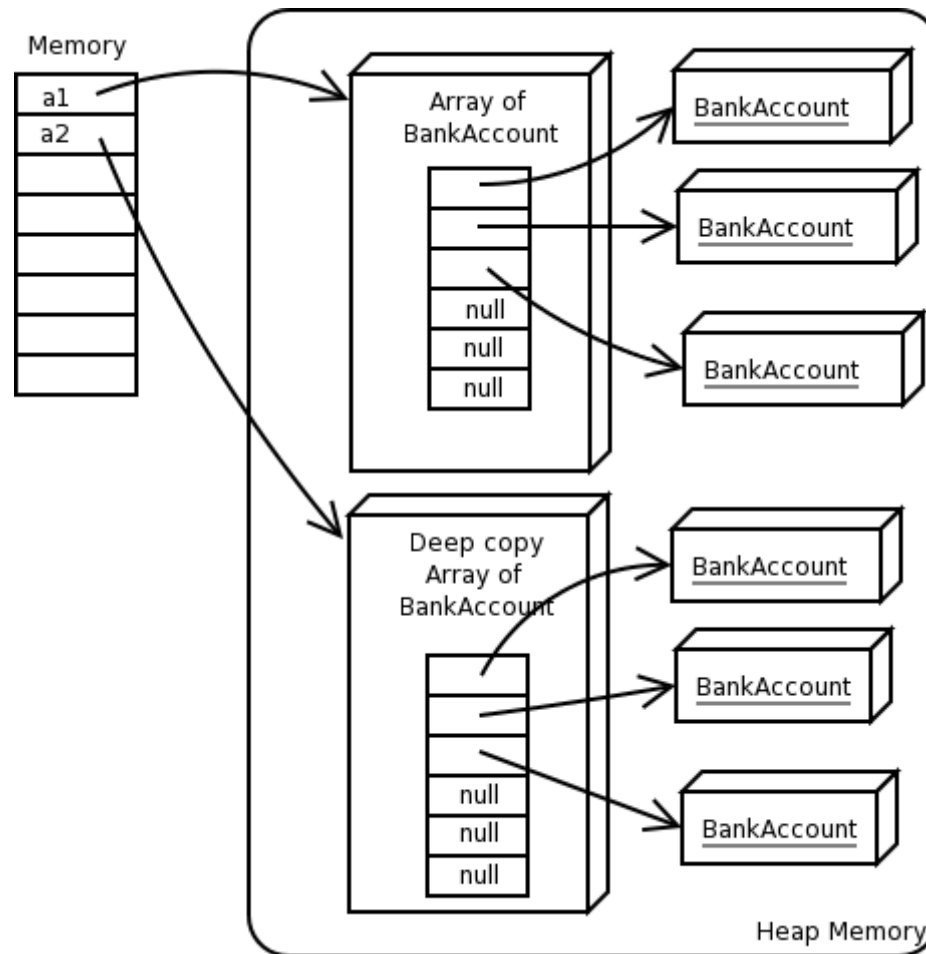
Deep Copying Example



Deep copying arrays

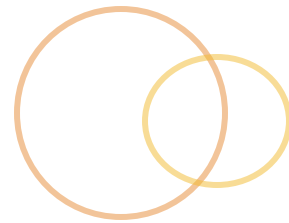
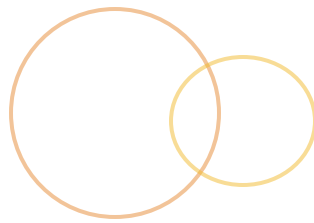
```
BankAccount a1 = new BankAccount[6];
for (int k = 0; k < 3; k++) {
    a1[k] = new BankAccount();
    // make a2 into a deep copy of a1 --
    a2 = new BankAccount[a1.length];
    for (int k = 0; k < a1.length; k++)
    {
        a2[k] = a1[k].clone();
    }
}
```

Deep Copying an Array



Result of deep copy

Summary



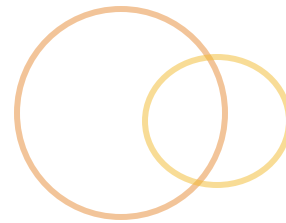
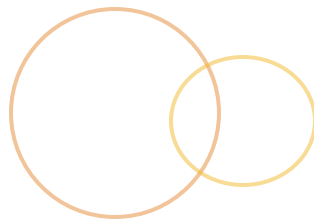
- ◎ Java has an Application Programming Interface (API) that provides a reference to all packages.
- ◎ There are common naming conventions that make reading and managing code easier.
- ◎ Java has a String Object that is immutable and can cause the memory heap to build.
- ◎ The StringBuilder object can be used in lieu of a String to manipulate Strings while avoiding heap growth.
- ◎ The Collections Framework is a flexible way to manage a collection of Objects.

Inheritance in Java

Introduction to Java



Objectives



- At the end of this module, you should be able to:
- Describe the OO concepts of abstraction and inheritance
- Use the **extends** keyword properly in Java to implement inheritance
- Understand overridden methods and constructors
- Define Encapsulation and how Java implements Encapsulation
- Use the final keyword effectively
- Use the **private** and **protected** keyword properly

Abstraction in the Real World - Concrete Classes



- ◎ Every object is of some type. This is how we naturally think about the world; it allows efficient information processing.
 - ◎ `String s1 = new String();`
 - ◎ `MyType mt = new MyType();`
- ◎ Types are defined by a process of abstraction or generalization--grouping a collection of objects together based on some common features; and creating a prototype

Abstraction

- Types exist only because the observers define them.
- During the design process all types in the problem domain are synthesized down to a set of design classes that are implemented in code.
- No "right" collection of classes exist:
 - The collection must address the business problem
 - Some classes are for design or implementation reasons
 - Different sets display different benefits, especially during maintenance
 - Your designs will probably get more elegant with experience

Inheritance in the Real World



Look at our banking example:

SavingsAccount
accountNumber balance interestRate
queryBalance(): deposit(amount): withdraw(amount):

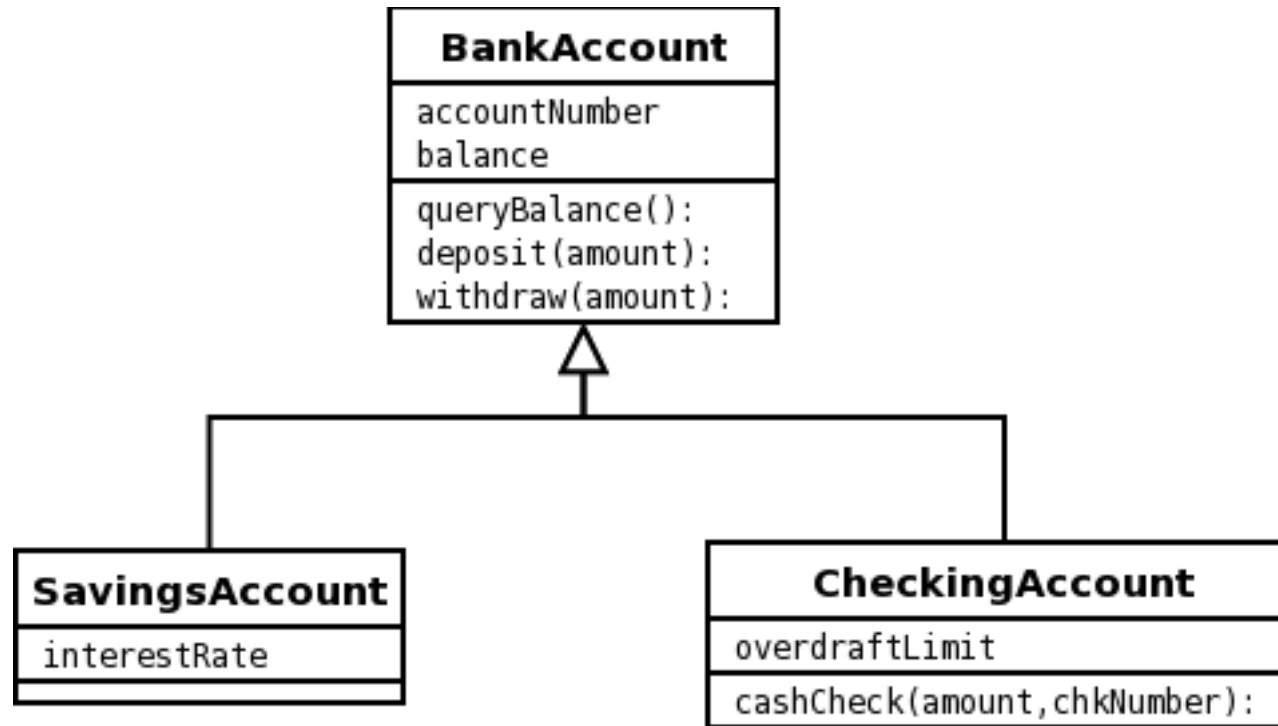
CheckingAccount
accountNumber balance overdraftLimit
queryBalance(): deposit(amount): withdraw(amount): cashCheck(amount, chkNumber):

The two bank account types

When doing abstraction analysis, look for

- common attributes
- common behaviors

Abstraction and Inheritance



Abstract BankAccount

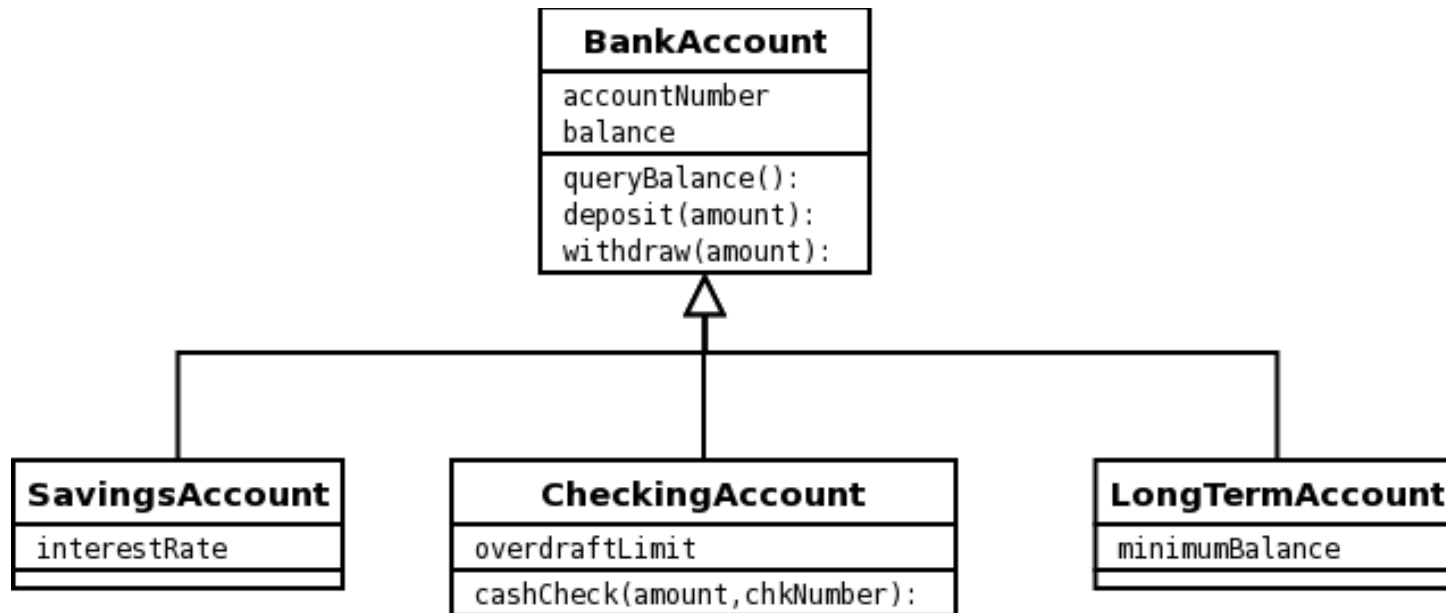
Notice where the commonality has been placed, in another class!

Abstraction and Inheritance (cont.)



- ⦿ "What is the relationship between personal customers and checking accounts?"
- ⦿ Reason by inheritance - Customers have accounts; therefore, personal customers can have accounts.
- ⦿ Checking account is a kind of account. It inherits the property of "being held" by a customer.

Abstraction and Inheritance (cont.)



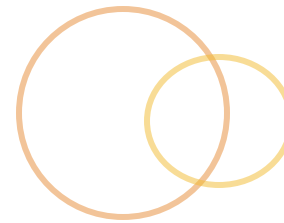
Adding a LongTermAccount

Some Jargon



- The BankAccount class is called
 - Base class, super class, parent class, or
 - The “generalization”
- The SavingsAccount and CheckingAccount classes are called
 - Derived classes, subclasses, child class, or
 - “Specializations”
- Different people have different terms which can be somewhat confusing

Inheritance in Java



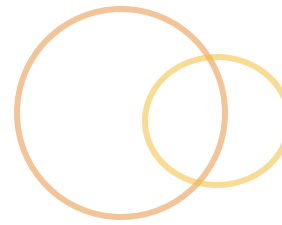
- ◎ Java only supports single inheritance:
 - ◎ A class can have only one parent
 - ◎ Follows along the lines of scientific classification
 - ◎ This is different from C++ and C#
 - ◎ Easier to manage
 - ◎ Less error prone – “no dreaded diamond of death”
- ◎ Inheritance in Java uses the `extends` keyword, i.e., class `Child` `extends` `Parent`.

Inheritance in Java (cont.)



- Inheritance in Java refers to
 - Instance variables
 - Instance methods
- Constructors are not inherited
- Static members are not inherited
- Static and non-static initializers are not inherited

Inheritance Example

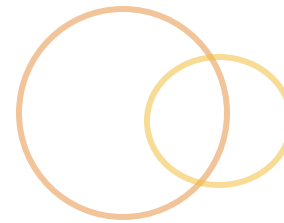


```
// Superclass BankAccount
class BankAccount {
    float balance;
    String accountNumber;
    float queryBalance() { /* code */
    float deposit(float amount) { /* code */
    float withdraw(float amount) { /* code */
}

// Subclass SavingsAccount
class SavingsAccount extends BankAccount {
    float interestRate;
}

// Subclass CheckingAccount
class CheckingAccount extends BankAccount {
    float overdraftLimit;
    float cashCheck(float amount, int ChkNumber) {
        /* code */
    }
}
```

Encapsulation



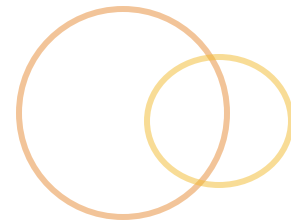
- ⦿ When something is encapsulated it is protected
- ⦿ In OOP we can encapsulate our classes, methods and variables
- ⦿ Encapsulation is an important mechanism that supplies programmers with tools to guarantee certain parts of a program will never change
- ⦿ There are many mechanisms in Java that will allow a program to protect sensitive areas of code

Benefits of Encapsulation



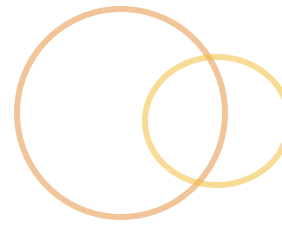
- Programmer defined access to classes, variables and methods
- Easier mechanism for testing Java code with unit test
- Ability to change parts of a program without it effecting other parts of the program
- Writing new requirements are faster and easier to manage
- Code is more flexible

Design Patterns



- Many design patterns use the concept of encapsulation
- Factory pattern encapsulates the making of objects
- Singleton pattern encapsulates and therefore controls a single instance of an Class

Access Modifiers



- Access modifiers define where a class, variable, or method can be accessed
- There are 4 standard access modifiers
- “final” is a nonstandard access modifier used to prevent changes to classes, methods and variables
- Access modifiers are used to implement encapsulation
- We will delve deeper in access modifiers later in this course

Final Classes



- ⦿ Declare a class to be final using the `final` keyword preceding the class definition.
- ⦿ This prohibits the class from being used as the base class in any inheritance structure
- ⦿ The reason for declaring a class final is always because of a design issue in the application
- ⦿ For example, we may declare a class final because having subclasses would allow objects to circumvent business rules or security checks

Final Class Example



```
// Superclass BankAccount is now final
// This will NOT compile because of the subclassing
final class BankAccount {
    float balance;
    String accountNumber;
    float queryBalance() { /* code */
    float deposit(float amount) { /* code */
    float withdraw(float amount) { /* code */
}

// Subclass SavingsAccount
class SavingsAccount extends BankAccount {
    float interestRate;
}

// Subclass CheckingAccount
class CheckingAccount extends BankAccount {
    float overdraftLimit;
    float cashCheck(float amount, int ChkNumber) {
        /* code */
    }
}
```

Private Access Modifier



- ⦿ The private access modifiers limits the scope of a variable or method to the enclosing class
- ⦿ Private data is not inherited, which means that sub classes do not have access to the **private** variables in the super class
- ⦿ Using the private access modifier is another example of **Encapsulation**
- ⦿ By declaring a method or variable as private, we are guaranteed no other program will be able to access and or harm these members
- ⦿ If it is a private variable we can supply access to that private variable with a public method

Private Access Modifier Inheritance Example



```
// BankAccount declares balance as private forbidding changes
// by any other
// part of the program including subclasses
class BankAccount {
    private float balance;
    /* -- more code -- */
}
// balance is a private variable and not allowed to be
// accessed by any other part
// of the program including subclasses.
// This will NOT compile because SavingsAccount is trying to
// access the private float balance
class SavingsAccount extends BankAccount {
    public queryBalance() {
        return balance;
    }
}
```

Protected Access Modifier



- ⦿ Use of the `protected` keyword for instance variables may not be desirable
 - ⦿ Protected members can be accessed by other objects of other class types within the same package
 - ⦿ Protected members can be accessed by children, even if the child resides in another package
- ⦿ If you need more “protection” than `protected` but not as restrictive as `private`, consider using the default access modifier (no modifier – not the `default` keyword)
 - ⦿ Accessible by subclasses in the same package
 - ⦿ Accessible by other classes in the same package
 - ⦿ Referred to as “package friendly”

Protected Access Modifier Inheritance Example



```
// balance is protected - okay now
class BankAccount {
    protected float balance;
    /* -- more code -- */
}
// Sub class SavingsAccount can access
// protected variable balance in super class
class SavingsAccount extends BankAccount {
    public queryBalance() {
        return balance;
    }
}
```

Implementing Inheritance



- Sometimes it is useful for a child to change an inherited behavior.
- This can be performed using method overriding.
- Instead of the inherited method being invoked, the overridden method will be invoked.
- In Java, the *type of the actual instance* determines the behavior you get, *not the type of the reference variable used to invoke the method*.
 - Called virtual invocation or late binding

Implementing Inheritance Example



```
class Parent {
    private String priVar = "(Parent Private)";
    protected String proVar = "(Parent Protected)";
    public String pubVar = "(Parent Public)";
    public void meth() {
        System.out.println("(Parent method)");
        System.out.println(priVar+" "+pubVar+" "+proVar);
    }
    public void methPar() {
        System.out.println("Parent method");
        System.out.println(priVar+" "+pubVar+" "+proVar);
    }
}

class Child extends Parent {
    public String pubVar = "(Child Public)";
    private String priVar = "(Child Private)";
    @Override // Ask the compiler to check we spelled it right
    public void meth() { //overridden method
        System.out.println("Child method");
        System.out.println(priVar+" "+pubVar+" "+proVar);
    }
}
```

Implementing Inheritance Example (cont.)



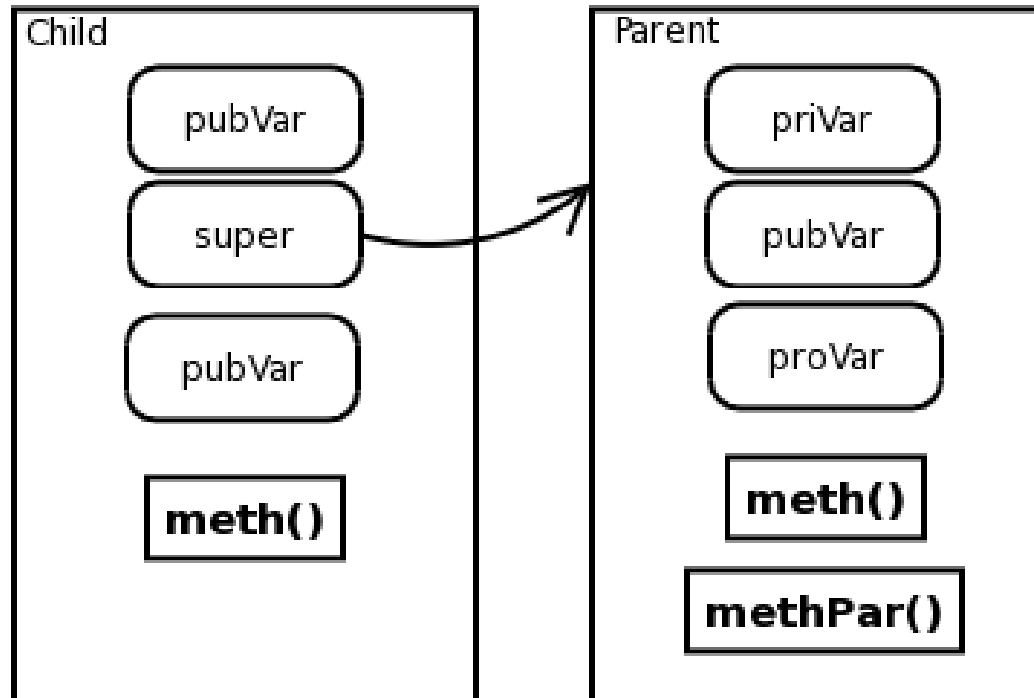
```
public static void main(String [] args) {  
    Child c = new Child();  
    c.meth();  
    c.methPar();  
}  
/* Output is  
* Child method  
* (Child Private)    (Child Public)  (Parent Protected)  
* Parent method  
* (Parent Private)   (Parent Public)  (Parent Protected)  
*/  
}
```

More About Implementing Inheritance



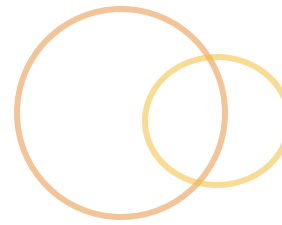
- ⦿ A child might need to interact with its parent
 - ⦿ This can be performed using a built-in reference `super`
 - ⦿ Use the dot-notation with `super`
- ⦿ Typically `super` is used to access an overridden method in the parent class.
- ⦿ It can also be used to explicitly access “shadowed” variables in the parent class.
- ⦿ Parent classes can encapsulate sensitive data and behaviors:
 - ⦿ Mark them `private`
 - ⦿ This might prevent method overriding

More About Implementing Inheritance (cont.)



Result of inheritance from example

Shadowing Example



```
class Parent {
    /* just like example
}
class Child extends Parent {
    public String pubVar ="(Child Public)";
    private String priVar = "(Child Private)";
    public void meth() {
        System.out.println("Child method");
        System.out.println(priVar+ " " +
                           super.pubVar + " " + proVar);
    }
    public static void main(String [] args) {
        Child c = new Child();
        c.meth();
        c.methPar();
    }
    /* Output is
    * Child method
    * Child Private)    (Parent Public)    (Parent Protected)
    * Parent method
    * Parent Private)    (Parent Public)    (Parent Protected)
    */
}
```

Invoking A Parent Method Example



```
class Parent {
    /* just like example
}
class Child extends Parent {
    public String  pubVar ="(Child Public)";
    private String  priVar = "(Child Private)";
    public void meth() {
        System.out.println("Child method");
        System.out.println(priVar+" "+super.pubVar+" "+proVar);
    }
public void up() {
    System.out.println("Up method");
    super.meth();
}
    public static void main(String [] args) {
        Child c = new Child();
        c.up();
    }
    /* Up method
    * Parent method)
    * Parent Private)    (Parent Public)    (Parent Protected)
    */
}
```

Extending and Implementing Methods



- ⦿ When overriding a method, the method signature in the child class can have the same access modifier
- ⦿ The method signature in the child class can be less restrictive

```
class Parent {  
    protected void m1() {}  
    public void m2() {}  
}
```

```
class Child extends Parent {  
    // this is allowed public is less restrictive  
    public void m1() {}  
    // not allowed, must be public  
    protected void m2() {}  
}
```

Accessing Super Class



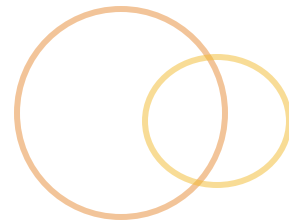
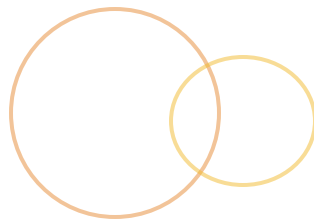
- Objects are initialized using constructors.
- JVM calls the constructor for each super class.
- By default, the JVM will call the default or no-argument constructor in the super classes when creating the child object.
- In some cases, the creation of the child object will require a different constructor be called in the parent class:
 - Use the `super` keyword in a manner similar to the use of the `this` keyword in the constructor
 - `super` must be the first execution in the constructor

Accessing Parent Constructors Example



```
class Parent {
    Parent() {
        System.out.println("Parent()");
    }
    Parent(int i) {
        System.out.println("Parent(int)");
    }
}
class Child extends Parent {
    Child () {
        super(); //redundant.. Default constructor would be called
        anyway
        System.out.println("Child()");
    }
    Child (int i) {
        super(i);
        System.out.println("Child(int i)");
    }
    Child (int i, int j) {
        super(i);
        System.out.println("Child(int i, int j)");
    }
}
```

Summary



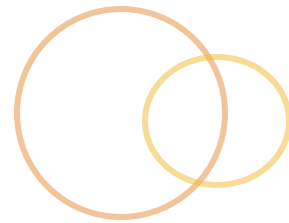
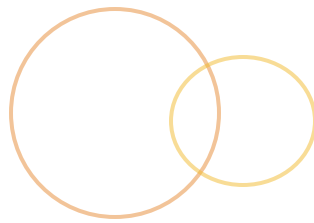
- ◉ We covered:
 - ◉ The OO concepts of abstraction and inheritance
 - ◉ Implementing inheritance through **extends** and overriding
 - ◉ Encapsulation
 - ◉ Using the final modifier
 - ◉ Using the private and protected keywords

Abstracting the Details

Abstract Classes

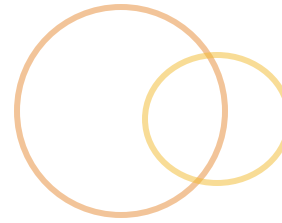


Objectives



- ② Understand abstraction
- ② Use abstract methods and classes
- ② Distinguish between abstract and concrete classes
- ② Declare abstract methods to create abstract classes

Abstract Classes



According to OxfordDictionaries.com, the definition of abstract is:

ab·stract

adjective

adjective: **abstract**

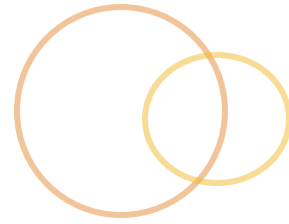
ab'strakt, 'ab, strakt/

1. existing in thought or as an idea but not having a physical or concrete existence.

"abstract concepts such as love or beauty"

Abstract classes are a way to provide inheritance and force behavior

Abstraction - Review



- Types exist only because the observers define them.
- During the design process all types in the problem domain are synthesized down to a set of design classes that are implemented in code.
- No "right" collection of classes exist.
 - The collection must address the business problem
 - Some classes are for design or implementation reasons
 - Different sets display different benefits, especially during maintenance
 - Your designs will probably get more elegant with experience

Abstract Classes

- Types are defined by a process of abstraction or generalization--grouping a collection of objects together based on some common features--and creating a prototype.
- Abstract classes can be used as a guarantee that any subclasses will inherit all of the desired state and behavior of the superclass.
- Abstract classes are also used as helper classes when some methods are going to be implemented and some left to the subclass.

Abstract Classes Provide Inheritance and a Guarantee



- ◎ Abstract classes implement some or even most of a solution.
- ◎ Many super classes implement as much of the work as possible.
- ◎ Some methods logically belong in a super class but can not be logically implemented.
- ◎ Labeling a class as abstract requires that the class be sub classed and finished off.

Abstract Classes are Declared “abstract”



- Abstract Classes in Java are always declared `abstract`.
- Abstract classes do not necessarily contain any abstract methods.

```
public abstract class Account { }
```

Abstract Classes May Not Be Instantiated



- Abstract classes are destined to be super classes.
- Abstract classes may not be instantiated.
- Any class with an abstract method must be declared as abstract.
 - We will talk about interfaces in the next module

Abstract Classes Have Constructors, Methods and Variables



- Abstract classes are like any other class and start with one implicit constructor

```
public AbstractClass() {  
    super();  
}
```

- Like normal Java classes, the default constructor may be overridden and new constructors are allowed.
- Abstract classes can contain all of the typical Java members: variables, implemented methods, inner classes, interface implementations, etc.

Good Practices

- It is a good practice to define an abstract class as a base class to be implemented and defined by sub classes.
- Many abstract super classes are used to implement most common methods of an Interface, and by doing so declare a type to be used later.
- Java.util.AbstractList is a great example of an abstract class used to implement some methods but not all:

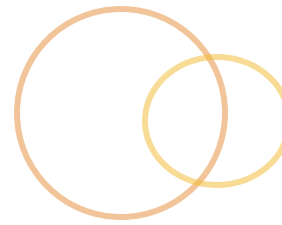
```
public abstract class AbstractList extends  
AbstractCollection implements List ...
```


Abstract Classes are the Opposite of Final Classes



- Abstract classes may never be labeled as final.
- Inheritance is a must for abstract classes and can not be used in any other way.
- An abstract class implies incompleteness, whereas a final class implies completeness.

Collecting Classes



- It is common to create an abstract super class as a means of being able to collect sub classes.
- A collection of sub classes can be used to realize polymorphism:

```
List<Account> accounts = new  
ArrayList<Account> ();  
accounts.add(new ChckingAccount());  
accounts.add(new SavingsAccount());  
accounts.add(new StockAccount());  
for( Account a: account)  
    a.getInterestRate();
```

What about Static Methods?

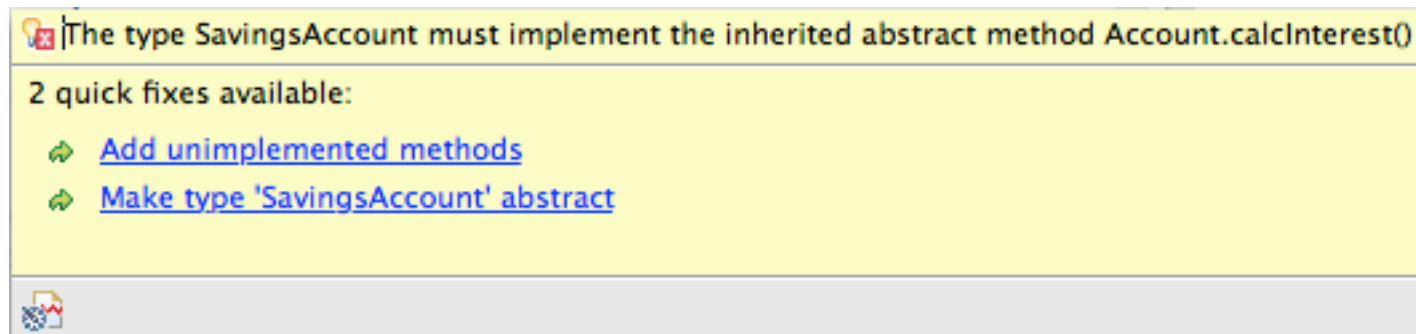


- Static methods are allowed in abstract classes.
- Static methods are not recommended because they cannot be overridden.

The Java Compiler Forces Assimilation



- Sub classes that extend abstract super classes must implement methods or be declared as abstract.
- Eclipse will give you the option:



Abstract Methods Have No Body



- Abstract methods have no method body:

```
public void accrueInterest();
```

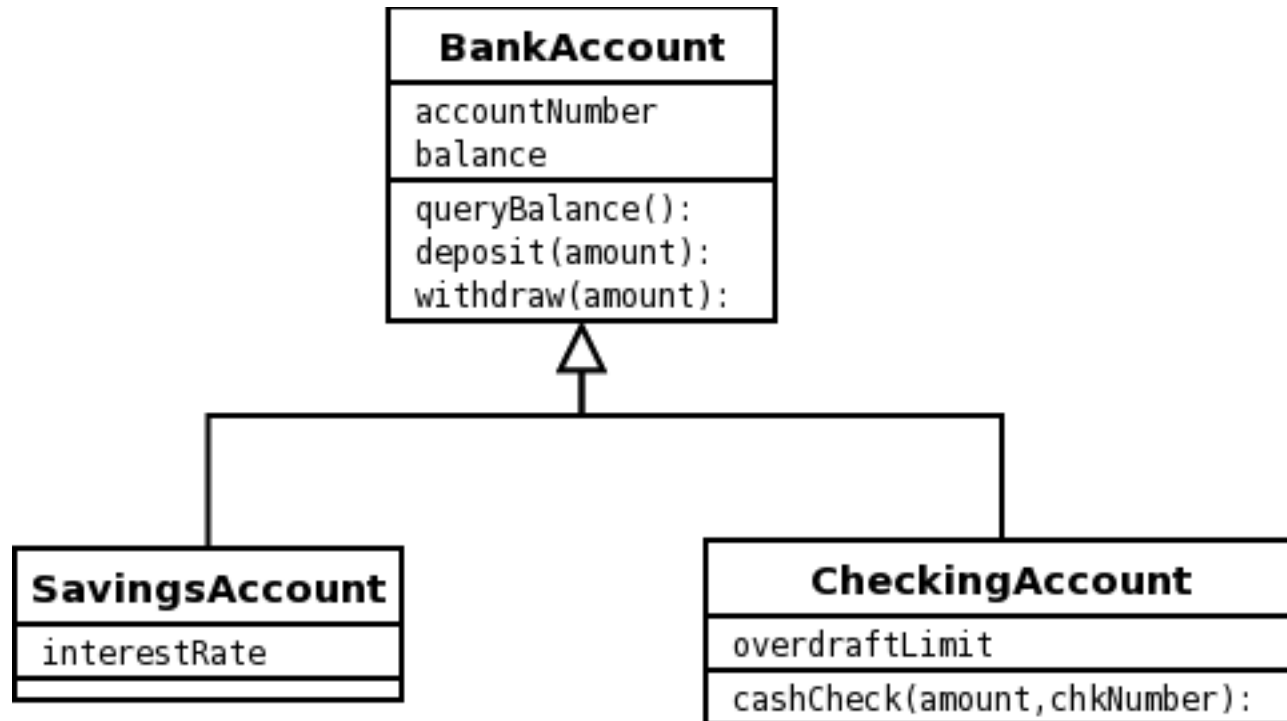
- Abstract methods have a method signature.
- Any class with an abstract method must be declared as `abstract`.
 - We will talk about interfaces in the next module
- Any class that extends an abstract class must implement all abstract methods or also be declared as `abstract`.

Abstracting the Bank Account



- ⦿ All accounts must capable of certain operations:
 - ⦿ Create, deposit, withdraw, calculate interest.
 - ⦿ Any deposit is OK.
 - ⦿ Withdrawals and transfers must have sufficient funds deposited or an overdraft agreement.
 - ⦿ Withdrawals accumulate over time.
- ⦿ All of these behaviors must be accomplished in some way.

Abstract BankAccount

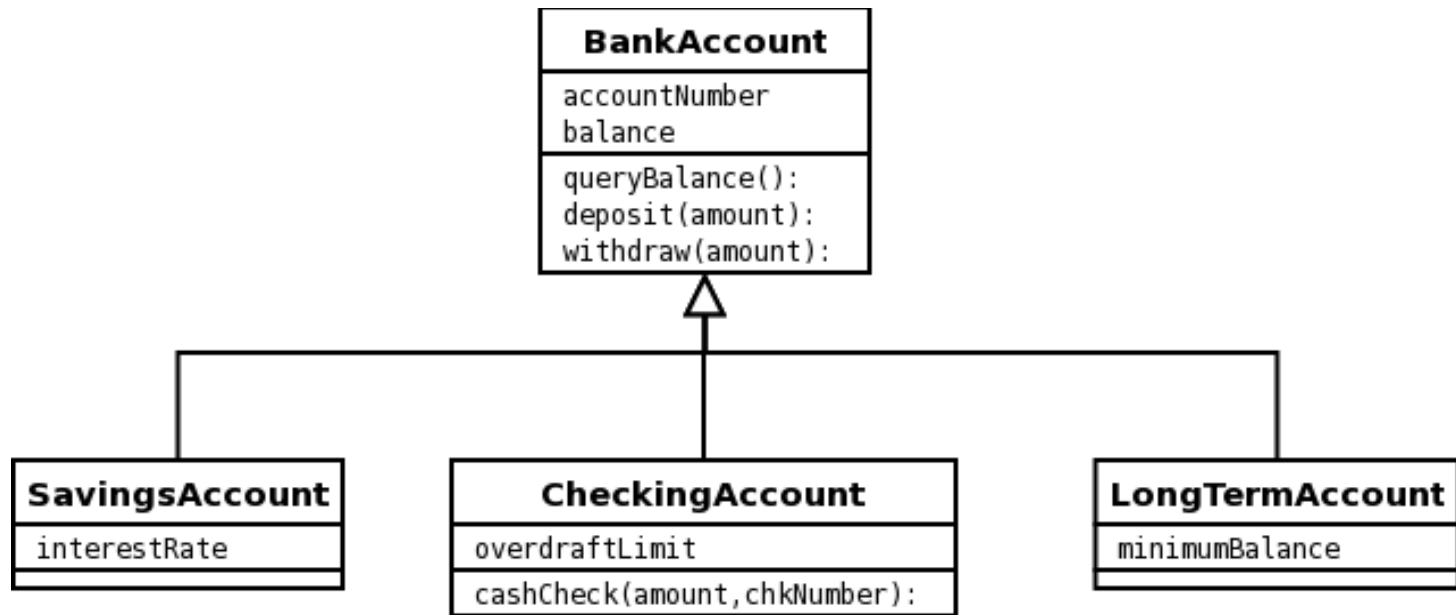


Abstract the Super Class Account



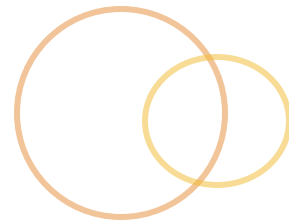
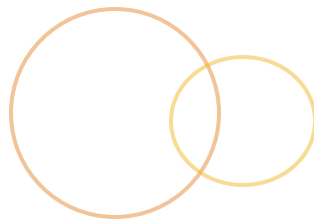
- The Account class is a generic class in that there will never be a plain old account class.
- Many accounts have the same variables and methods up to a point.
- Abstract the Account class to contain all of the common implemented methods and variables.
- Provide abstract methods for behavior that must reside in sub classes.

Abstraction and Inheritance (cont.)



Adding a LongTermAccount

Summary



In this module, we covered:

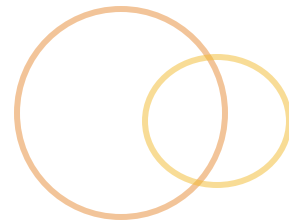
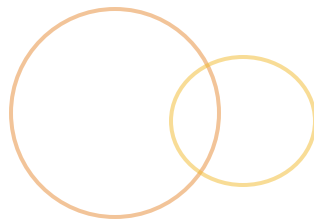
- 🕒 Understanding abstraction
- 🕒 Using abstract methods and classes
- 🕒 Distinguishing between abstract and concrete classes
- 🕒 Declaring abstract methods to create abstract classes

Behavioral Inheritance With Interfaces

Introduction to Java

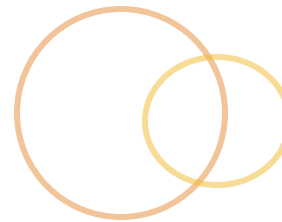
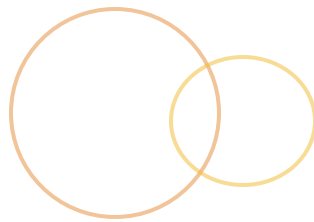


Objectives



- ◎ Define interfaces
- ◎ Understand multiple inheritance
- ◎ Use interfaces
- ◎ Recognize inherited abstract methods
- ◎ Understand object-oriented patterns
- ◎ Use up-casting and down-casting with interfaces

Interfaces



- Interfaces are defined in .java files.
 - Structure similar to classes and compiler creates .class files

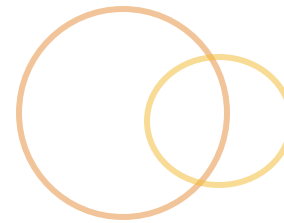
```
[public] interface <interface_name> {  
    [...]  
}
```

- Interfaces may be empty or may contain:
 - public abstract method declarations
 - public static final fields (constants)
- Abstract methods are:

- Instance, *not class*, methods without a body

```
[public] abstract void doSomething();
```

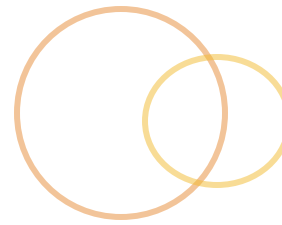
Interfaces (cont.)



- ⦿ A class can take on the behavior of an interface by:
 - ⦿ Implementing the interface, using `implements` *and*
 - ⦿ Defining all of the dictated abstract methods
- ⦿ Classes inherit nothing from interfaces.
- ⦿ Interfaces are a “contract to perform.”
- ⦿ Classes can implement many interfaces:

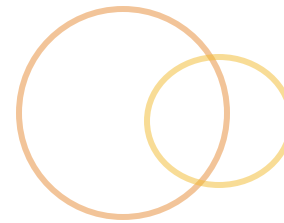
```
class Child implements Interface1,  
    Interface2, Interface3
```

Multiple Inheritance



- ⦿ A class can implement as many interfaces as the programmer desires.
- ⦿ Some programmers define this as multiple inheritance.
- ⦿ Interfaces are state free and only define behavior.
- ⦿ Interfaces define types.

Interface Example



```
public interface LedgerAccount {  
    public abstract String credit(String amt);  
    public abstract String debit(String amt);  
}
```

```
public class BankAccount {  
    public String deposit(String amt) { /*...*/ }  
    public String withdraw(String amt) { /*...*/ }  
}
```

```
public class SavingsAccount extends BankAccount  
    implements LedgerAccount {  
    public String credit(String amt) { /*...*/ }  
    public String debit(String amt) { /*...*/ }  
}
```

```
public class LoanAccount implements LedgerAccount {  
    public String credit(String amt) { /*...*/ }  
    public String debit(String amt) { /*...*/ }  
}
```


Multiple Interface Example

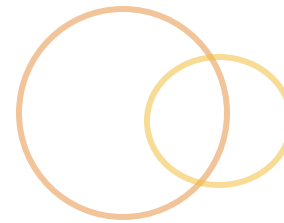


```
public interface LedgerAccount {  
    public abstract String credit(String amt);  
    public abstract String debit(String amt);  
}
```

```
public interface Persistent {  
    public abstract void read(String url);  
    public abstract void write(String url);  
}
```

```
public class SavingsAccount extends BankAccount  
    implements LedgerAccount, Persistent {  
    public abstract String credit(String amt) { /* ... */ }  
    public abstract String debit(String amt) { /* ... */ }  
    public abstract void read(String url) { /* ... */ }  
    public abstract void write(String url) { /* ... */ }  
}
```

Type Polymorphism



- Polymorphism allows an object to be viewed as different types:
 - SavingsAccount is also:
 - A BankAccount
 - A LegerAccount
 - A Persistent
 - A `java.lang.Object`

Polymorphism Example



```
interface LedgerAccount {
    /* body code */
}

interface Persistent {
    /* body code */
}

class BankAccount {
    /* body code */
}

class SavingsAccount extends BankAccount implements LedgerAccount,
Persistent {
    /* body code */
}

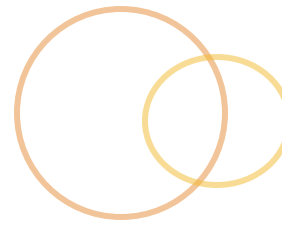
// Elsewhere in code...

SavingsAccount s1 = new SavingsAccount();

LedgerAccount s2 = new SavingsAccount();

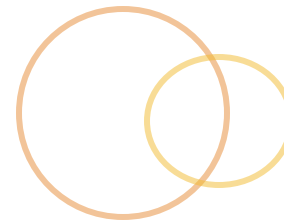
BankAccount s3 = s1;
```

Interfaces Follow the Policy Pattern



- Policy pattern is also known as the Strategy Design pattern
- Allows runtime to choose correct implementation
- Polymorphic behavior
- If we have 10 types of accounts, an interface that defines a withdrawal can be called and the correct implementation for that object will be executed

Type Casting



- If an object has more than one type, how do you determine its functionality?
 - Look at the type of the reference variable
 - Utilize “casting”
- Casting converts the type of a reference
 - Might lose access to specific functionality
 - Reference example

```
s1 = (SavingsAccount) s2;
```
 - Java handles widening (up-casting) automatically
 - Narrowing (down-casting) must be performed manually
 - Recall that casting also works with primitives

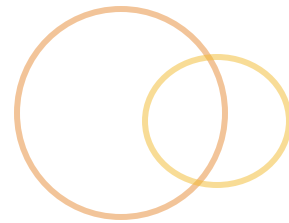
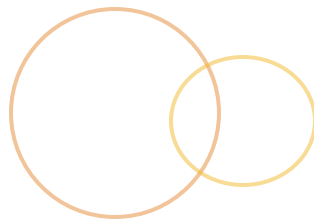
Type Casting Example



```
interface LedgerAccount {
    /* code */
}
interface Persistent {
    /* code */
}
class SavingsAccount extends BankAccount implements LedgerAccount,
Persistent {
    /* code */
}
```

```
// Elsewhere in code...
SavingsAccount s1 = new SavingsAccount();
BankAccount s3 = s1; // this is Okay
s1 = s3;    //illegal - compiler error
s1 = (SavingsAccount) s3; // now this is okay
LedgerAccount s4 = new SavingsAccount(); // okay
s1 = (SavingsAccount) s4; // need to cast here too
```

Summary



We covered:

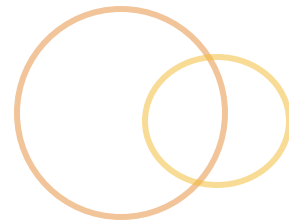
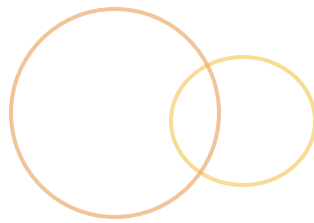
- 🕒 Interface definition
- 🕒 Multiple inheritance
- 🕒 Using interfaces
- 🕒 Inherited abstract methods
- 🕒 Object-oriented patterns
- 🕒 Using up-casting and down-casting with interfaces

Working With Methods

Introduction to Java



Objectives



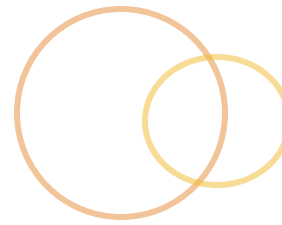
- ◎ The Call Stack
- ◎ Understanding method signatures
- ◎ Defining methods
- ◎ Naming methods
- ◎ Overloading methods

Java Methods



- ◎ Methods are how objects behave
- ◎ Equivalent to functions
- ◎ Methods define what a class can do
- ◎ Interfaces and abstract classes have methods with no body
- ◎ Methods need a body to execute behavior

Method Call Stack



- ⦿ Java programs exist as a stack of method calls
- ⦿ Each Java process has its own stack of methods
- ⦿ The stack of methods is called the call stack or sometimes the execution stack
- ⦿ When no more methods are left to call, the program ends

The Six Parts of a Method



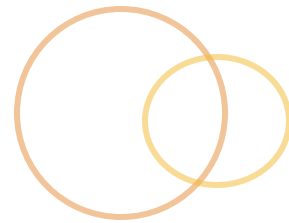
- ⦿ Modifier
- ⦿ Return type
- ⦿ Name
- ⦿ Parameter or argument
- ⦿ Exception declaration
- ⦿ Method body

Methods Can Have Many Forms



```
package com.auto_owners.examples;  
public class MethodModifiers {  
public static void main(String[] arg)  
{  
  
public int method1() throws Exception{return  
0;}  
  
protected void method2(int x) {/* method  
body */}  
  
static void method3(float x) {/* method body  
*/}  
  
}
```

The Modifier



- Defines scope
- Who can call the method
- Calling a method adds it to the call stack
- Methods without an access modifier assume default access
- Default access is package-private
 - Only another object or classes within the package can access method

Bytecode of Modifiers



Compiled from "ReturnValues.java"

```
class com.auto_owners.examples.ReturnValues {  
    com.auto_owners.examples.ReturnValues();
```

Code:

```
    0: aload_0  
    1: invokespecial #8          // Method java/lang/Object."<init>":()V  
    4: return
```

```
static int method1();
```

Code:

```
    0: iconst_2  
    1: ireturn
```

```
static void method2(int);
```

Code:

```
    0: getstatic      #18          // Field java/lang/System.out:Ljava/io/PrintStream;  
    3: new            #24          // class java/lang/StringBuilder  
    6: dup  
    7: ldc            #26          // String 2+  
    9: invokespecial  #28          // Method java/lang/StringBuilder."<init>":(Ljava/lang/String;)V  
   12: iload_0  
   13: invokevirtual  #31          // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;  
   16: ldc            #35          // String =  
   18: invokevirtual  #37          // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;  
   21: invokestatic  #40          // Method method1:()I  
   24: iload_0  
   25: iadd  
   26: invokevirtual  #31          // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;  
   29: invokevirtual  #42          // Method java/lang/StringBuilder.toString:()Ljava/lang/String;  
   32: invokevirtual  #46          // Method java/io/PrintStream.println:(Ljava/lang/String;)V  
   35: return
```

```
public static void main(java.lang.String[]);
```

Code:

```
    0: bipush        14  
    2: invokestatic  #55          // Method method2:(I)V  
    5: return
```

}

Return Value of a Method

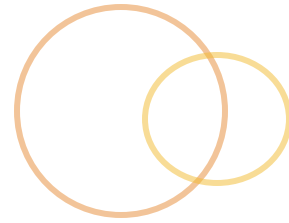
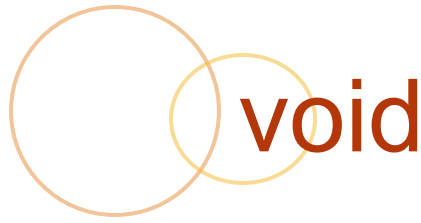
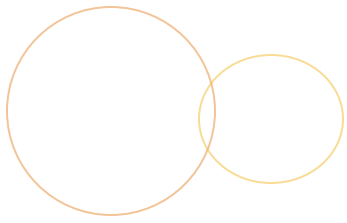


- Methods are called from within other methods, starting with the main method.
- When the method is finished executing the code inside the body of the method, it returns to the calling method.
- In some circumstances the method call ends abruptly, in these cases we return a different kind of Object called an Exception.
- Methods must declare that they may return an exception is the throws clause of the method.

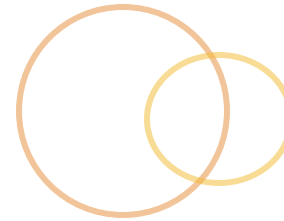
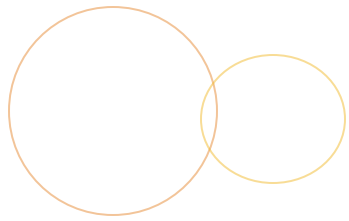
Methods Must Declare a Return Value



- ⦿ Every method has to have a return value associated with it.
- ⦿ The return value must be either a primitive data type or a declared reference data type (an object).
- ⦿ Sometimes a method doesn't need to return anything, in which case the return type **void** is used to indicate “nothing returned.”



- Some methods perform some function and then return to the calling code with nothing to report.
 - In such a case we use the void return type.
- Void simply means that no message will be returned from the method.
- Methods that return void are typically methods that accept an argument and are used as mutators to alter some state.



- The way to return from a method is by using the keyword “return.”
- If a method returns “void,” meaning nothing, we don’t have to use the “return” keyword.
- We can use “return” without an argument by simply calling “return” within the method body.
- When we call return, the method execution ends and control returns to the calling method.

public int someMethod()



- If a method does declare a return value other than void, then it is under contract to return that value.
- A method must return a value that is of the same type as the return clause.
- One caveat is that if the method stipulates the return of an object reference and the object is not available, the method could return the null value to complete the contractual agreement.

Completing the Contract

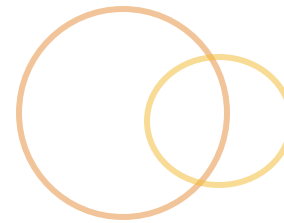


```
public SomeObject getSomeObject()
{
    if(someObject != null && someObject.exist())
        return someObject;
    else
        return null;
}
```

This is valid but could cause problems if the program is not expecting a null to come back.

JavaDocs should warn of this situation.

Looking for null



```
public void useSomeObject()
{
    //Here we call the method getSomeObject() expecting
    //a SomeObject reference to be returned

    SomeObject so = getSomeObject();

    //Now we check to make sure that we have a valid //Object to reference before using the
    //returned SomeObject

    if(so != null)
        so.sayHello();
    else
        System.out.println("Can't SayHello object is null");
}
```

Method Signatures and Overloading



- ◎ The signature of a method is the method name plus the list of parameters.
- ◎ All method signatures in a class must be unique, which means that either:
 - ◎ Two methods have different names

OR

 - ◎ Two methods have the same name, but different argument lists.

Method Signature are Scoped to a Class



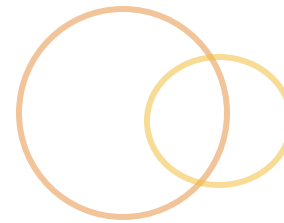
- ⦿ The uniqueness of signatures only applies within a class definition.
- ⦿ Methods can have the same signatures if they are in different classes.
- ⦿ As an analogy, think of a method signature as being similar to a phone number, and the class definition as analogous to an area code.
- ⦿ The phone numbers can be the same as long as the area codes are different.

Overloading Doesn't Make Code Run Faster



- Overloading does not make code run faster or better, but it makes life easier for the programmer.
- Overloading gives the programmer options of how to call a method.
- Overloading allows for better code reuse.
- This will become clearer when we talk about constructor methods and initializing objects.
- Overloading provides a way to call the same method and pass it different arguments.

Calling a Method



- Methods are called within a class by invoking the method signature in code.
- The compiler knows which of the overloaded methods to call because it looks at the whole signature when making the determination.
- Overloading gives the programmer options of how to call the method.
- Return values are not part of a signature.

Java is a Strongly Typed Language



- ◎ Java is considered a strongly typed language.
- ◎ This means that you have pass an argument that is the exact type of the parameter being asked for.
- ◎ There is some wiggle room when an object is passed
 - ◎ as an argument
 - ◎ and the object has multiple types based on hierarchy and interfaces
 - ◎ and can be cast into another type of object.

Argument Promotion

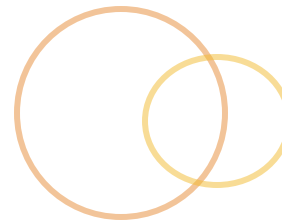
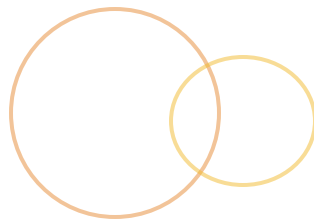


- ⦿ When the compiler encounters a method call, it first tries to find a method where the signature matches exactly.
- ⦿ If there is not exact match, the compiler tries to find a signature that would match if the argument were *promoted* or converted to the parameter type in the signature.

Arguments Are Allowed to Grow



- ◎ Java allows primitive values room to grow.
- ◎ This means that a 32-bit float primitive type is allowed to automatically grow to a 64-bit double primitive type.
- ◎ Java will not allow it to happen the other way around, however.
- ◎ The compiler tries to find a signature that would match if the argument were *promoted* or converted to the parameter type in the signature



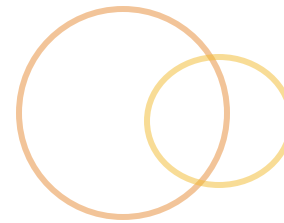
- The Java API lists all of the packages, classes, methods, variables, interfaces, enums, etc. included in the JDK.
- Knowing what methods can be called and what to pass the methods is crucial to programming in Java.
- This is one of the main reasons behind the JavaDoc tool:
 - If we create a program and want to make that program available to other programmers, they have to know exactly how to call each method we make available.

Method Parameters as Local Variables



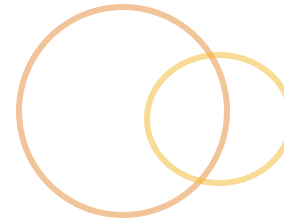
- ⦿ The enclosing curly braces define methods.
- ⦿ When an argument is passed into a method to satisfy the parameter list, that argument is only valid inside those curly braces.
- ⦿ This is the scope of the variable.

Cut and Paste



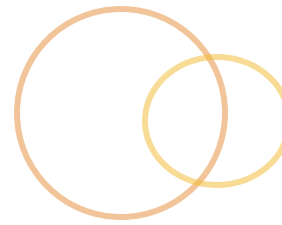
- One of the practices that is *not* recommended for Java programmers is to cut and paste the whole method body.
- The rule of thumb is not to duplicate code.
- One method should call another method and fill in the gaps.
- This way there will be one method where most of the work is done--it is typically the method with the most parameters--and then other overloaded methods will fill in where they can.

Overload Example



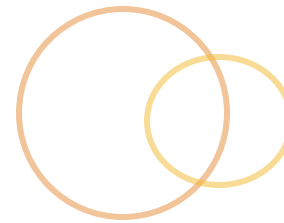
```
public class OverloadExample {  
    public int count(int i1, int i2, int i3, int i4)  
    {  
        return i1+i2+i3+i4;  
    }  
    public int count(int i1, int i2, int i3)  
    {  
        return count(i1, i2, i3, 0);  
    }  
    public int count(int i1, int i2)  
    {  
        return count(i1, i2, 0, 0);  
    }  
    public int count(int i1)  
    {  
        return count(i1, 0, 0, 0);  
    }  
}
```

The Main Method



- In Java every program begins in the main method.
- The main method is responsible for creating objects, by calling their constructor methods, and then calling other methods.
- This is done when the Java Virtual Machine creates a process, called a thread, and then builds a stack of methods to be executed in order.

The Main Method



```
package com.auto_owners.examples;  
public class ArgumentPromotion {  
    public static void main(String [] args) {
```

- ◎ This is a public method, meaning anyone can access it.
- ◎ If it was not public, then the Java Virtual Machine could not call this main method.
- ◎ It is static, which means it can be called right from the class. The JVM would call:

```
com.auto_owners.examples.ArgumentPromotion.main(args[]);
```

The Main Method Returns void



- ⦿ We are not allowed to return anything to the JVM, so it always has to return void.
- ⦿ The name of the method is main, which makes up the first half of the method signature and the JVM will always call main.
- ⦿ Since Java is case sensitive, main must be all lowercase.

Passing in Arguments to Main

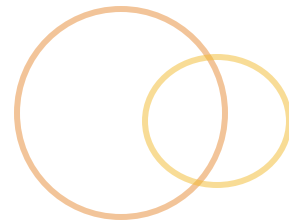
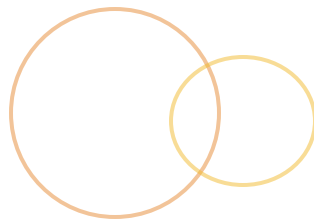


- The main method always takes an array of strings as the argument.
- This array can be passed in at the command line in the form of:

```
java com.auto_owners.examples.ArgumentPromotion  
    argument1 argument2
```

- or we can use Eclipse and right click the class that contains the main that we want to run and select Run As → Run Configurations.

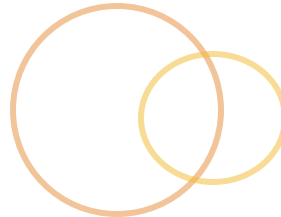
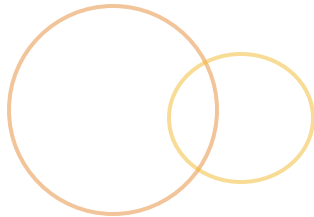
Summary



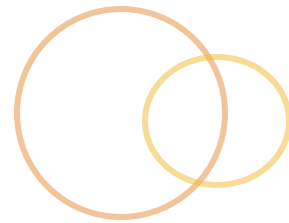
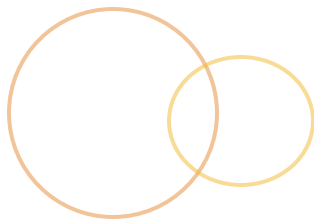
We covered:

- ⦿ Six parts of a method
- ⦿ Defining methods
- ⦿ Modifiers
- ⦿ Return value
- ⦿ Calling a method
- ⦿ Naming methods
- ⦿ Overloading methods

Constructors and Visibility

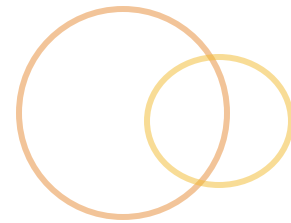


Objectives



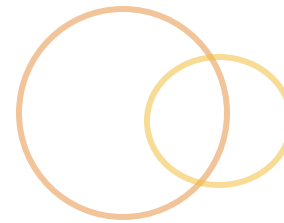
- At the end of this module, you should be able to
 - Understand how the **new** operator uses memory
 - Overload constructors to customize object creation
 - Describe how constructors build all objects in the class hierarchy
 - Use access modifiers
 - Build encapsulation using access modifiers
 - Privatize data to apply business rules for access

Constructors



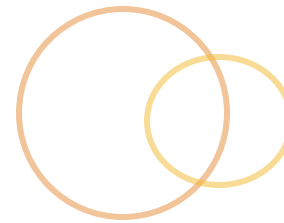
- Objects are created through a `new SomeType()` call
- After the memory has been created, the object is initialized in the constructor
- Constructors may be thought of as initialization methods
 - Note there is no return value

Constructor Purpose



- The purpose of the constructor is to initialize the newly created Objects.
- Providing a class constructor:
 - Allows correct instance variable initialization
 - Any other initialization or startup code can be executed
 - Perform complex initialization logic that can not be done as an explicit initialization, e.g. loops
- Constructors allow us to call the new keyword
 - What follows the keyword **new** must match the signature of a constructor
- If you provide no constructors, compiler provides one
 - Referred to as the default constructor
 - No arguments
 - There is no behavior specified in the default constructor

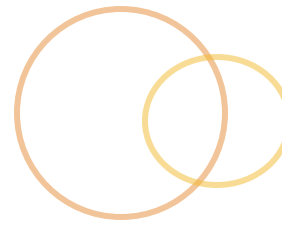
Constructor Rules



Constructors must abide by some specific rules

- ⦿ The constructor *always*
 - ⦿ Has the same name as the class
 - ⦿ Is case sensitive
- ⦿ Constructors can be overloaded
 - ⦿ Similar to method overloading
 - ⦿ May be multiple constructors with different argument lists
- ⦿ Constructors do not declare a return value
 - ⦿ This is not the same as returning `void`
 - ⦿ A constructor initializes the newly created object

Constructor Example



```
class BankAccount {
    float balance = -1.0F;
    String accountNumber = "NotSet";
    int accountStatus = -1;
    char accountType = " ";

    BankAccount(String num, char type) {
        accountNumber = num;
        accountType = type;
        balance = 0.0F;
        accountStatus = (type == 'p')? 100: 0;
    }
    BankAccount(String num, char type, float bal) {
        accountNumber = num;
        accountType = type;
        balance = bal;
        accountStatus = (type == 'p')? 100: 0;
    }
    BankAccount(String num) {
        accountNumber = num;
    }
    /* -- rest of class -- */
}
```

Proper Constructor Form



- Typically a class defines multiple constructors
 - Each constructor varies by argument list
 - Though the constructors are different, they should perform the same level of initialization
- Having many constructors
 - Provides flexibility
 - Can be error prone if done wrong
- Constructors can refer to other constructors
 - To minimize redundant code
 - Provide centralized initialization
 - Simplify maintenance

Proper Constructor Form (cont.)



- ◉ When referring to other constructors
 - ◉ Utilize a built-in mechanism - `this (...)`
 - ◉ Think of `this (...)` as a constructor calling another constructor
 - ◉ Works like a method call
 - ◉ JVM determines which constructor to call
- ◉ Use `this (...)`
 - ◉ As the first execution in your constructor
 - ◉ Can perform other operations once `this (...)` “returns”

Proper Constructor Form



```
class BankAccount {
    float balance = -1.0F;
    String accountNumber = "NotSet";
    int accountStatus = -1;
    char accountType = " ";

    BankAccount(String num, char type, float bal) {
        accountNumber = num;
        accountType = type;
        balance = bal;
        accountStatus = (type == 'p')? 100: 0;
    }
    BankAccount(String num, char type) {
        this(num, type, 0.0F);
    }
    BankAccount(String num) {
        this (num, 'p');
    }
    /* -- rest of class -- */
}
```

The Default Constructor



In the first module, we used the disassembler (`javap`) to look into our `HelloWorld` class

```
Compiled from "HelloWorld.java"
public class HelloWorld {
    public HelloWorld();
        Code:
            0: aload_0
            1: invokespecial #1          // Method java/lang/Object."<init>":()V
            4: return

    public static void main(java.lang.String[]);
        Code:
            0: getstatic     #2          // Field java/lang/System.out:Ljava/io/PrintStream;
            3: ldc           #3          // String Hello World
            5: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
            8: return
}
```


The Default Constructor



```
//This compiles and runs
class Test1 {
    int x;
    public static void main(String [] args) {
        Test1 t = new Test1(); // this is the default constructor
        System.out.println(t);
    }
}

//This does not compile
class Test2 {
    int x;
    // Adding this constructor prevents the default
    // constructor is not provided
    Test2(int xs) {
        x = xs;
    }
    public static void main(String [] args) {
        Test2 t = new Test2(); // this constructor no longer exists.
        System.out.println(t);
    }
}
```

The Instance_INITIALIZER



- ◎ Initializers are invoked prior to constructors
- ◎ Multiple initializers are permitted, they are executed from top to bottom of the class
- ◎ The syntax is simply an unadorned block in the class

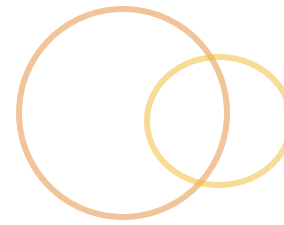
```
class ThingOne {  
    int someNumber;  
    {  
        someNumber = (int)(Math.random() * 1000);  
        if (Math.random() > 0.9) { someNumber = 0; }  
    }  
    /* Rest of class definition */  
}
```

Heap Allocation

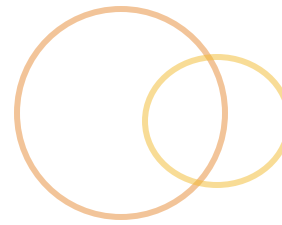


- ⦿ The JVM allocates memory for objects
- ⦿ Each object that is built contains variables and methods from each class in the hierarchy
- ⦿ JVM manages objects in memory, allocating memory when **new** is called
- ⦿ Methods are stored on top of memory space and variables are stored on the bottom

Java Memory Map

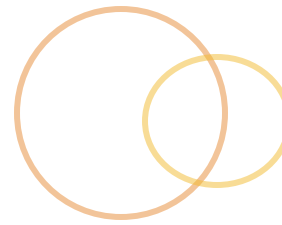


Access Modifiers and Encapsulation



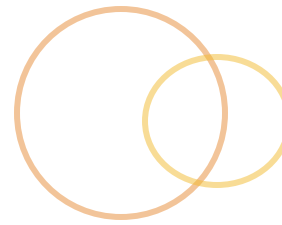
- Object-Oriented Analysis and Design encourages the use of encapsulation
- Encapsulation is defined as data hiding
 - Think of encapsulation as a black box
 - Hides the “dirty” or sensitive details of objects
 - Prevents misuse
 - Thwarts “hacking”
 - Encapsulation should be applied to objects
- Objects are logical containers of
 - Data or state
 - Functionality or behavior

Access Modifiers and Encapsulation (cont.)



- ⦿ An object's variables should not be exposed outside the object
 - ⦿ Instance variable exposure can allow direct variable access
 - ⦿ This would circumvent any business rules you have in place
 - ⦿ Would also allow object to obtain and corrupt sensitive data
- ⦿ Sensitive and critical behaviors should also be hidden from other objects

Access Modifiers and Encapsulation (cont.)



Access modifiers to create encapsulation

- Access modifiers define a level of accessibility for classes
 - Class variables
 - Class methods
- Access modifiers define a level of accessibility for instances
 - Instance variables
 - Instance methods
 - Constructors
- Basic syntax is:

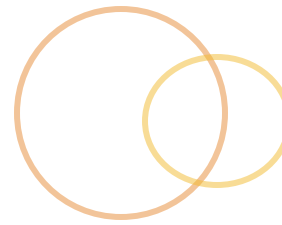
- Variables

```
<access_modifier> Type identifier;
```

- Methods

```
<access_modifier> <return_type> identifier(<parameter list>)
```

Access Modifiers and Encapsulation (cont.)



- ◎ Java has four access modifiers:
 - ◎ `private` – only the class and object can access
 - ◎ *default* – only class, object, and subclass in same library (*package*)
 - ◎ `protected` – the class, object and subclasses in any package
 - ◎ `public` – any class, object, subclass
- ◎ We will cover access modifiers in more detail when we discuss packages
- ◎ The default access modifier is automatically added if an access modifier is not explicitly specified

Private Variable Example



```
class BankAccount {  
    // Instance Variable  
    private float balance;  
    // Constructor  
    BankAccount() {  
        balance = 0.0F;  
    }  
    // Instance Methods  
    float queryBalance() {  
        return balance;  
    }  
    float withdraw(float amt) {  
        if ((amt > 0.0F) && (amt <= balance)) {  
            balance -= amt;  
        }  
        return balance;  
    }  
    float deposit(float amt) {  
        if (amt > 0.0F) {  
            balance += amt;  
        }  
        return balance;  
    }  
}
```

Private Variable Example (cont.)



```
class BankApp {  
    public static void main(String [] args) {  
        System.out.println("Starting banking application...");  
        // Create a bank account  
        BankAccount act = new BankAccount();  
        act.deposit(100.00F);  
        System.out.println("Balance is " + act.queryBalance());  
        // This following line will not compile !!  
        System.out.println("Balance is "+ act.balance);  
        System.out.println("Ending banking application...");  
    }  
}
```

//produces the compiler error output

BankApp.java:9: balance has private access in BankAccount

```
System.out.println("Balance is "+ act.balance);  
                                ^
```

1 error

Private Variable Example



```
class BankAccount {
    // Instance Variables
    String accountNumber = null;
    char accountType;
    float balance;
    int accountStatus;
    // Instance Methods
    float queryBalance() {
        return balance;
    }
    float withdraw(float amt) {
        if ((amt > 0.0F) && (amt <= balance)) {
            if ((accountType == 's' && accountStatus == 100) ||
                (accountType == 'c' && accountStatus == 0)) {
                balance -= amt;
            }
        }
        return balance;
    }
    . . .
}
```

Private Variable Example (cont.)



```
float deposit(float amt) {  
    if (amt > 0.0F) {  
        if ((accountType == 's' && accountStatus == 100) ||  
            (accountType == 'c' && accountStatus == 0)) {  
            balance += amt;  
        }  
    }  
    return balance;  
}
```

Private Method Example



```
class BankAccount {  
    // Instance Variables  
    String accountNumber = null;  
    char accountType;  
    float balance;  
    int accountStatus;  
    // Private Instance Methods  
    private boolean isAccountOK() {  
        return ((accountType == 's' && accountStatus == 100) ||  
                (accountType == 'c' && accountStatus == 0));  
    }  
    // Instance Methods  
    float queryBalance() {  
        return balance;  
    }  
}
```

. . .

Private Method Example



```
float withdraw(float amt) {  
    if ((amt > 0.0F) && (amt <= balance)) {  
        if (isAccountOK()) {  
            balance -= amt;  
        }  
    }  
    return balance;  
}  
float deposit(float amt) {  
    if (amt > 0.0F) {  
        if (isAccountOK()){  
            balance += amt;  
        }  
    }  
    return balance;  
}  
}
```

Private Variable Example



Private access is permitted between objects of the same type

```
class A {  
    private int var = 0;  
    void changeVar( A otherA) {  
        otherA.var++;  
    }  
    public static void main(String [] args) {  
        // create an two A objects  
        A firstA = new A();  
        A secondA = new A();  
        // use the first A object to change the private data in  
        // the second A object  
        firstA.changeVar(secondA);  
    }  
}
```

Public Method Example



Public Access

```
class BankAccount {  
    // Instance Variables  
    String accountNumber = null;  
    char accountType;  
    private float balance;  
    int accountStatus;  
    // Private Instance Methods  
    private boolean isAccountOK() {  
        return ((accountType == 's' && accountStatus == 100) ||  
                (accountType == 'c' && accountStatus == 0));  
    }  
    // Public Instance Methods  
    public float queryBalance() {  
        return balance;  
    }  
}
```

. . .

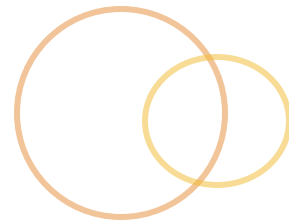
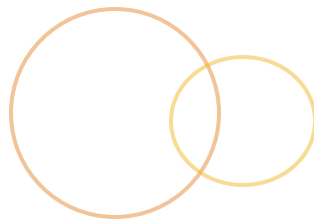
Public Method Example (cont.)



Public Access (continued)

```
public float withdraw(float amt) {  
    if ((amt > 0.0F) && (amt <= balance)) {  
        if (isAccountOK()) {  
            balance -= amt;  
        }  
    }  
    return balance;  
}  
  
public float deposit(float amt) {  
    if (amt > 0.0F) {  
        if (isAccountOK()) balance += amt;  
    }  
    return balance;  
}  
}
```

Summary

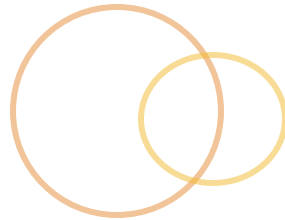


We covered

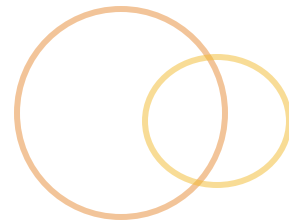
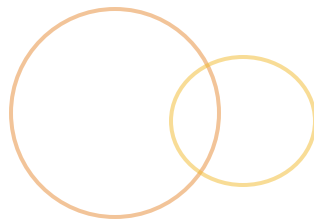
- ⦿ The **new** operator and how it uses memory
- ⦿ Overloading constructors to customize object creation
- ⦿ How constructors build all objects in the class hierarchy
- ⦿ Access modifiers
- ⦿ Encapsulation using access modifiers
- ⦿ Privatization of data to apply business rules for access

Variables, Operators and Data

Part I



Objectives



At the end of this module you should be able to:

- ☉ Understand the rules for creating legal variable names in Java
- ☉ Describe and use the basic primitive data types in Java
- ☉ Determine the data type of a literal

Strong Typing in Java



- ◎ Java is a strongly typed language
 - ◎ Each variable and each expression has a type
 - ◎ Can be identified by the compiler at compile time
 - ◎ A variable's type cannot be changed
- ◎ In loosely typed languages, like JavaScript & VB

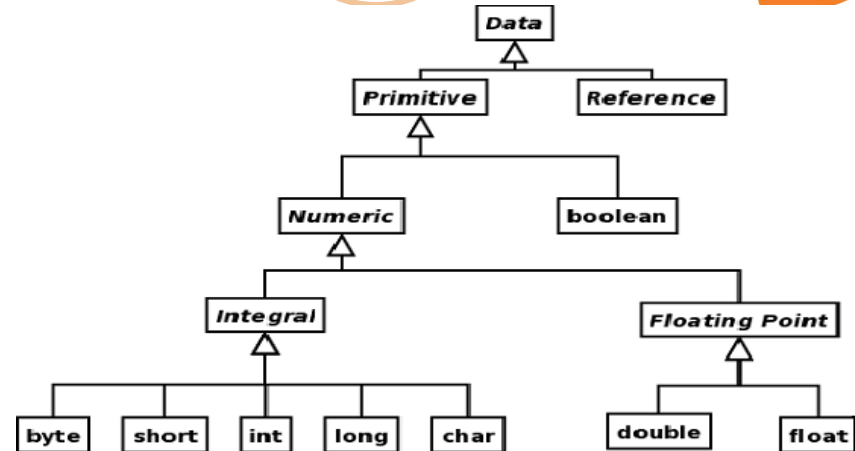
```
// JAVASCRIPT: This is not allowed in JAVA!!!  
// Declare a variable "x" with no type  
var x  
x = "Hi there"    // x is holding string data  
x = 1234          // x is now holding numeric data  
y = x + "343"    // String or numeric operation??
```

- ◎ Strong typing helps prevent errors

Data Types in Java



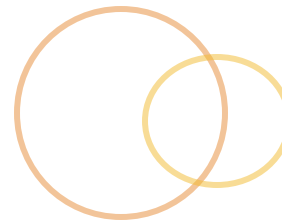
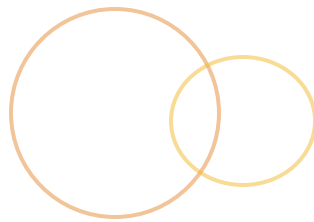
- There are two types
 - Reference data
 - Primitive data



Hierarchy of data types in Java

- The primitive data types resemble the types in C / C++
- Data types in Java are defined by the language specification
 - They are platform independent
 - For example, the data type `int` is *always* four bytes long

Identifiers



- ⦿ Identifiers are used to name
 - ⦿ Classes (interfaces, enums, and annotations)
 - ⦿ Variables (object fields and local variables)
 - ⦿ Methods
- ⦿ Identifier rules are platform independent
 - ⦿ Arbitrarily long sequence of letters and digits
 - ⦿ Case sensitive
 - ⦿ The first character must not be a digit
 - ⦿ Any valid letter in the Unicode character set
 - ⦿ Underscore "_" and dollar sign "\$" are also permitted
 - ⦿ Must **not** contain any white space
 - ⦿ Must **not** be the same as reserved Java keywords

Reserved Keywords

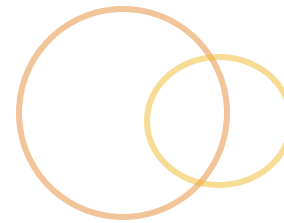


abstract	continue	for	new	switch
assert	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const*	float	native	super	while
true	false	null		

*goto **and** const are reserved but not used

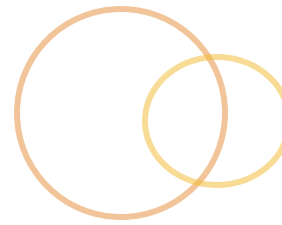
true, false, **and** null are reserved literal names

Variable Names



Account_Balance	<i>// valid - remember that _ is allowed</i>
\$34	<i>// valid - remember that \$ is allowed</i>
this	<i>// invalid - same as reserved keyword</i>
π	<i>// valid - Greek letter pi is a Unicode letter</i>
This	<i>// valid - different in case from 'this'</i>
Next.item	<i>// invalid - symbol "." not allowed</i>
23skidoo	<i>// invalid - must start with letter</i>

Declaring a Variable

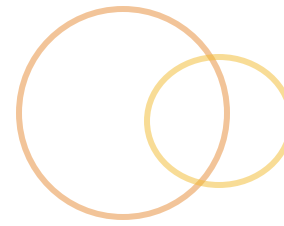


- Variables are declared in Java with the syntax

```
type name [= initializing_expression];
```

- It is good programming practice to *initialize variables when they are declared*.
- A variable can also be initialized after it is declared.
- Java will prevent the use of uninitialized local variables.

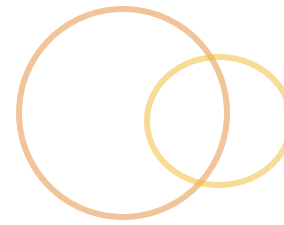
Declaring a Variable



```
String best = "Best"; // Preferred - initialized at declaration
String okToo;         // Declared - not initialized
int x;                // Declared - not initialized

okToo = "value";       // Now okToo is initialized.
x = x + 1;             // ERROR! Use of an uninitialized variable
```

Declaring a Variable



```
boolean a,b;                // a and b are both of type boolean
boolean c = true, d = false; // initialization for both c and d
boolean e,f = true;         // WARNING! Only f is initialized!
```

```
int var1;
boolean var2 = true;
var1 = 9
```

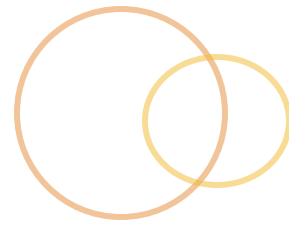
Positioning Variable Declarations



- ⦿ The basic principal in Java, and in OOP in general, is to declare a variable at its point of first usage
 - ⦿ This allows it to be initialized in its declaration

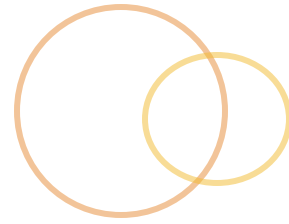
```
class Test {  
    public static void main(String [] args) {  
        int sum = 0;  
        for (int counter = 0; counter < 10; counter++)  
            sum = sum + counter;  
        String message = "The sum is " + sum;  
        System.out.println(message);  
    }  
}
```

Boolean Data Types



- `boolean` data is either `true` or `false`
- In Java, numeric and other variables are not `boolean`, and cannot be interpreted as such
 - In C, C++, JavaScript and others, zero is false, any defined nonzero value is true
- A `boolean` can only have the value `true` or `false`

Boolean Data Types



// THIS DOESN'T WORK IN JAVA, but in C++ you do can this:

```
int x = 43;  
// nonzero x is taken as a true  
if (x) {  
    printf("x is %d", x);  
}
```

// In Java we must have a boolean variable or expression.

```
int x = 43;  
boolean test = (x == 43);  
// boolean variable is OK  
if (test) {  
    System.out.println("x is "+x);  
}  
// OK because result of the == test is boolean  
if (x == 43) {  
    System.out.println("x is " + x);  
}
```

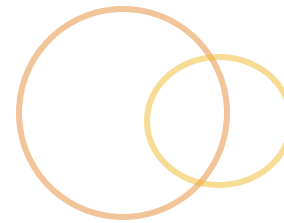
Integral Numeric Data Types



- Integral values are signed
 - char type has numeric properties, but is unsigned
- Integral values do not contain a decimal point

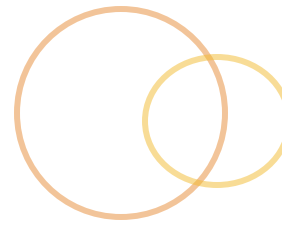
Type	Bytes	Minimum Value	Maximum Value
byte	1	-128	127
short	2	-32768	32767
int	4	-2147483648	2147483647
long	8	-9223372036854775808	9223372036854775807

Integral Literals



- ⦿ A sequence of digits without a decimal point is assumed to be integral data
 - ⦿ literals are of type `int` unless there is an `L` -- upper or lowercase -- immediately following the digits
 - ⦿ In this case, the literal is taken to be a `long`
 - ⦿ `7836` and `-98` are `int`
 - ⦿ `881L` and `-91121` are `long`
- ⦿ Literals are interpreted in base 10 unless
 - ⦿ The literal starts with a `0`, it is interpreted as base 8
 - ⦿ The literal starts with a `0x` or `0X`, it is interpreted as base 16

Integral Literals



```
63      // an int in base 10
-63     // a negative int in base 10
63L     // a long in base 10
063     // an int in base 8 (equal to 51 in base 10)
063L    // a long in base 8
-063L   // a negative long in base 8

091     // illegal!  Cannot have the digit 9 in base 8!

0x33    // an int in base 16 (equivalent to 51 in base 10)
0X33L   // a long in base 16
0xFF    // an int in base 16
0xff    // same as the previous line - case does not matter.
0xgl    // illegal! Can only have a-f as base 16 digits.
-0xFF   // a negative int in base 16
```

Floating Point Data Types



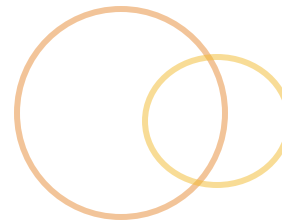
- Floating point are numerical values with fractional parts.
- Two kinds of floating point numbers:
 - `float`: 4 bytes long
 - Largest `float` is $3.4028234 \text{ E } +38$
 - About 6 or 7 significant digits
 - `double`: is 8 bytes long
 - Largest `double` in magnitude is $1.79769313486231570 \text{ E } +308$
 - About 15 significant digits

Infinites, Negative Zeros and Non-numbers



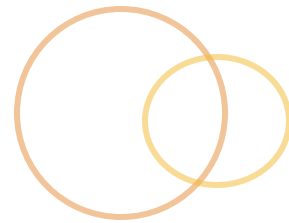
- ◉ `Double.POSITIVE_INFINITY`
- ◉ `Double.NEGATIVE_INFINITY`
- ◉ `Double.isInfinite(infinity)`
- ◉ `Double.NaN`
 - ◉ `double a = Double.NaN, b = Double.NaN;`
 - ◉ `a != b;`
 - ◉ But `a.equals(b) == true` to allow use in hash structures
- ◉ `-0.0 // negative zero`
- ◉ Also `Float.POSITIVE_INFINITY` etc.

Infinity Example



```
class InfinityTest {  
    public static void main(String [] args) {  
        // Set up bigd as a large double  
        double bigd = 1e306;  
  
        // loop - we should see bigd overflow about the third iteration  
        for (int i=1;  
            (i<100) && (bigd<Double.POSITIVE_INFINITY);  
            i++){  
            System.out.println("Iteration="+ i +": bigd="+bigd);  
            bigd = bigd * 10.0;  
        }//end for loop  
    }//end main  
}//end class
```

Floating Point Literals

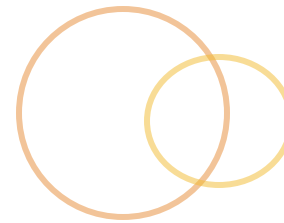


Floating point literals

```
38.0           // double
38.0f          // float
38.98D         // double
1.78e23        // double
1.78e23f       // float

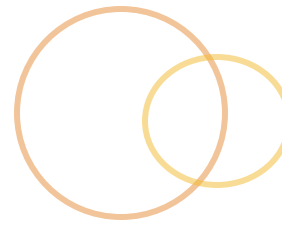
-789.983       // double
-1.89e-17F     // float
```

Character Data



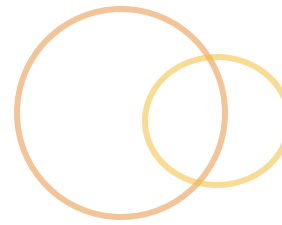
- ⦿ A kind of integral data
 - ⦿ The type name is `char`
 - ⦿ Takes on values from 0 to 65535
- ⦿ Java supports the Unicode standard - each character is stored as a two-byte representation
- ⦿ ASCII is a subset of Unicode - Java handles the ASCII/Unicode conversions behind the scenes

Character Literals



- Character literals usually represent a single Unicode character in single quotes.
- Character literals can also be the numeric code for Unicode characters:
 - Use Unicode escape sequence notation
 - `'\udddd'` where `dddd` is the hexadecimal representation of the Unicode character
- Certain common nonprintable characters, as well as the single and double quote and backslash, have special escape sequences that are recommended for use instead of the corresponding Unicode escape sequence.

Character Literals



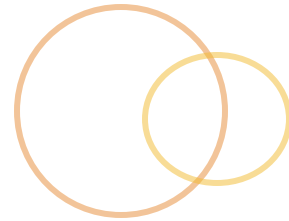
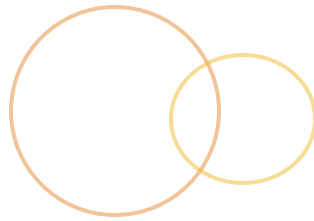
Character literals

<code>'a'</code>	<code>'7'</code>	<code>'ξ'</code>	<code>'©'</code>	<code>' '</code>
<code>'\''</code>	<code>'\\'</code>	<code>'\u0F34'</code>	<code>'\u0004'</code>	<code>'\ffd1'</code>
<code>'_'</code>				

`'ab'` // Not a char literal - two characters between quotes
`'\ug189'` // Not a char literal - illegal Unicode code.

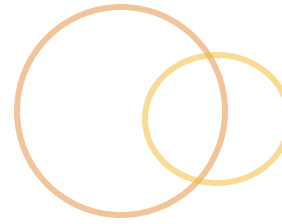
Unicode Escape Sequences

<code>'\b'</code>	<code>/* \u0008: backspace BS */</code>
<code>'\t'</code>	<code>/* \u0009: horizontal tab HT */</code>
<code>'\n'</code>	<code>/* \u000a: linefeed LF */</code>
<code>'\f'</code>	<code>/* \u000c: form feed FF */</code>
<code>'\r'</code>	<code>/* \u000d: carriage return CR */</code>
<code>'\"'</code>	<code>/* \u0022: double quote " */</code>
<code>'\''</code>	<code>/* \u0027: single quote ' */</code>
<code>'\\'</code>	<code>/* \u005c: backslash \ */</code>



- There is no string primitive data type in Java.
 - `Strings` are actually a reference data type that is implemented in the Java SE APIs.
 - Java allows `String` data to be used syntactically as if it were a primitive data type in many cases.
 - Intended to make working with character strings more "programmer friendly."
- `String` literals are sequences of Unicode characters.
 - Enclosed in double quotes
 - `char` escape sequences are valid for `String`

Chars and Strings



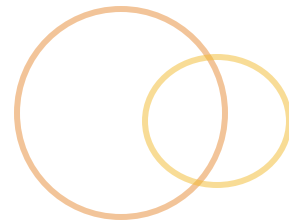
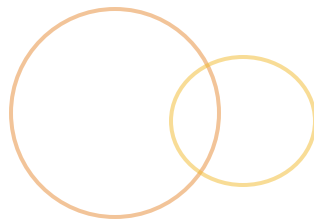
Using Character data

```
char a = 'a';           // single character
char b = 'b';           // single character
char nl = '\n';         // escape code for newline
char x = '\u7878';      // Unicode escape sequence
a + b;                  // the result is an int.
```

Strings

```
String s = "This is a string";
s = "This is a string with a backspace \b in it";
s = "This is a string with a \" double quote inside";
String t = s;
t = "";                // this is the empty string
t = 'a';               // illegal! 'a' is not a string.
```

Summary



We covered:

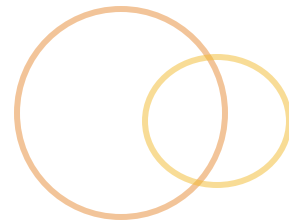
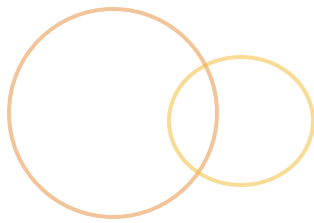
- ⦿ The rules for creating legal variable names in Java
- ⦿ Describing and using the basic primitive data types in Java
- ⦿ Using String data
- ⦿ Determining the data type of a literal

Variables, Operators, and Data

Part II



Objectives



At the end of this section you should be able to:

- 🕒 Describe operators and expressions
- 🕒 Describe how operators are used to create expressions
- 🕒 Describe the operators in Java, the kinds of data they operate on, and the types of expressions they produce
- 🕒 Describe the difference between narrowing and widening operators
- 🕒 Use the cast operator correctly

Operators and Expressions

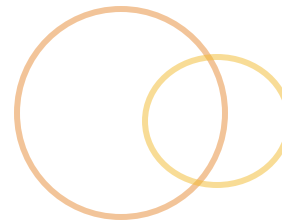
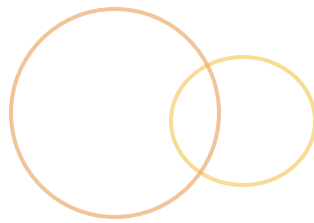


- ⦿ An expression in Java is something that evaluates to a result.
 - ⦿ A variable is an expression because it evaluates to a result (i.e., the value of the data it contains).
 - ⦿ A literal is also an expression.
- ⦿ An operator is used to combine two expressions to produce a new expression.
- ⦿ Think of variables as nouns and operators as verbs.
 - ⦿ Combine nouns and verbs to create phrases.
 - ⦿ Phrases correspond to expressions.

Operators and Expressions(cont.)

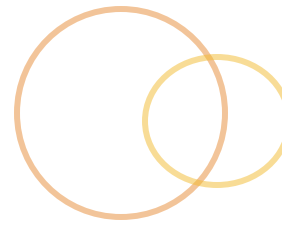
- ◎ Every expression has a type, just like variables.
- ◎ The type of an expression is determined by the type associated with the data results when we evaluate the expression.
- ◎ For example, a `boolean` expression is one that results in a `boolean` result, while a `String` expression is one that results in a `String`.

Operators



- Operators are of three valences in programming languages:
 - Unary operators - operate on a single expression
 - Binary operators - combine two expressions
 - Ternary operators - combine three expressions
- Most operators fall in the binary operator category
- There is only one ternary operator in Java
- Java does not allow operator overloading like in C++ / C#

Operators - Example



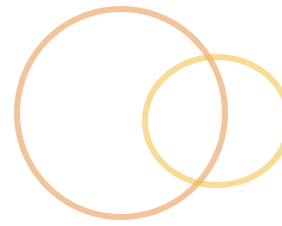
- There is no relationship between the type of operator (category) and the expression type.

Operators in Java

```
1 + 4 // arithmetic operator "+" operating on two ints
1.0 * 4.1 // arithmetic operator "*" operating on two doubles
true && false // logical operator operating on two booleans
true + false // illegal! - you can't do arithmetic on booleans
```

```
// Error in the following line, even though almost all the
// operators in the expression are arithmetic, the final result
// is a boolean, and cannot be assigned to x.
int x = (((34 + 12)/13) * (89-16)/(13 *2)) > 0;
```

Arithmetic Operators



- ◎ Java uses the standard $+$ $-$ $*$ $/$ $\%$ operators.
 - ◎ Java also has the increment and decrement operators.
 - ◎ Operators are defined for both the integral and floating point types.
- ◎ Mixed Mode Arithmetic
 - ◎ All arithmetic operators work on either two integral operands or two floating point operands.
 - ◎ Smaller operand types are promoted to the larger type, and to *at least* `int`.
 - ◎ If one operand is integral and one operand is floating point then the integral operand is converted to a floating point number before the operation takes place.

Mixed Mode Arithmetic



- Converting from integral to floating point values might produce a loss of precision.
- Binary representation of floating point values does not reliably match “exact” decimals.

Mixed Mode Arithmetic

```
1 + 4           // result is integral 5
1.1 + 4.2       // result is 5.3 (or 5.30000000000000001 sometimes)
1.0 + 4.0       // result is 5.0 - still floating point
1.0 + 4         // result is 5.0 - one operand is floating point
```

Division and Modulus



Division in Java

```
17 / 3          // result is 5 - integral division
17 % 3          // result is 2 - remainder of 17 / 3
17.0 / 3.0      // result 5.2 - floating point division
17.3 / 3        // result is 5.766666666666667
                // -- floating point division
17.0 % 3        // result is 2.0 - floating point modulus
14.5 % 3.32     // result is 1.22000000000000006
                // whatever that means.
```

Increment and Decrement



- Java provides increment and decrement operators: ++ and --
- Postfix: <VAR>++ and <VAR>--
 - The value of the variable is used in the expression first, then modified
- Prefix - ++<VAR> and --<VAR>
 - The value of the variable is incremented or decremented first, then modified

x++	is equivalent to	x = x + 1
++x	is equivalent to	x = x + 1
x--	is equivalent to	x = x - 1
--x	is equivalent to	x = x - 1

Increment and Decrement Example



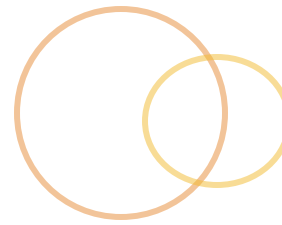
Increment and Decrement

```
int i = 23;
double d = 0.0;

// Print out i
System.out.println("Value of i is "+ i);
// Print out ++i - i is printed out after being incremented
System.out.println("Value of ++i is "+ (++i));
// Print out i again - it has been incremented.
System.out.println("Value of i is "+ i);

// Print out i
System.out.println("Value of i is "+ i);
// Print out i++ - i is printed out before being incremented
System.out.println("Value of i++ is "+ (i++));
// Print out i again - it has been incremented.
System.out.println("Value of i is "+ i);
// Just to see that it works with floating points
System.out.println("Value of ++d is "+ (++d));
```

Increment and Decrement Example Output



```
Value of i is 23  
Value of ++i is 24  
Value of i is 24  
Value of i is 24  
Value of i++ is 24  
Value of i is 25  
Value of ++d is 1.0
```

Output of the IncTest

Comparison Operators With Numerics



- Comparison operators are defined for numeric and character data.
- The result of all comparisons is either true or false.

Comparison Operators in Java

==	Equality
<	Less than
>	Greater Than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal

Comparison Operators With Other Types



- Only the `==` and `!=` operators are defined for `boolean` types.
- Not all comparison operators work for `String` types:
 - Remember, a `String` is not a primitive data type.
 - Only the `==` and `!=` work with `String` types, but not quite as you might expect.

Cautions with Comparison Operators



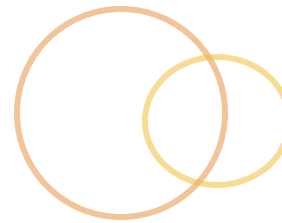
- ⦿ It is possible for a loss of precision to occur when working with floating point numbers.
 - ⦿ This is unavoidable.
 - ⦿ Sometimes using floating point numbers in comparisons can produce counterintuitive results.

Relational operators and floating point numbers

```
double d = 1.1 + 4.2; // As we saw, could be 5.30000000000000001
d == 5.3              // Because of representation, this is false
```

```
double d1 = 1e300;    // d1 is a very large number
d1 < (d1 + 1)         // because of representation, this is false
d1 == (d1 + 1)        // but this is true
```

Logical Operators



- The operators `&`, `|`, `^`, and `!` all work according to the usual rules of boolean operations.
- The two operators `&&` and `||` are called short circuit operators.

Evaluation of Logical Operators

Assume x and y are boolean expressions.

$x \ \& \ y$ *true if both x and y are true, false otherwise*

$x \ | \ y$ *false if both x and y are false, true otherwise.*

$x \ ^ \ y$ *true if either x or y is true but not both*

$!x$ *false if x is true, true if x is false*

$x \ \&\& \ y$ *same as `&`, but if x is false, y is not evaluated*

$x \ || \ y$ *same as `|` but if x is true, y is not evaluated.*

Short Circuit Evaluations



- In a short circuit evaluation, we stop evaluating the expression as soon as we know what the outcome will be.
- This avoids unnecessary processing if the first part of the evaluation is `false`.

Short circuit evaluation

```
int x = 0;
boolean test = false & (1 == ++x);
// x is incremented and is now 1
test = false && (2 == ++x);
// second operand is not evaluated!
// x still has value 1
```

Assignment Operators



- Java allows C/C++ style assignment operator notation

Operator Assignment

<code>x = x * 34;</code>	<code>//can be written as</code>	<code>x *= 34;</code>
<code>x = x / 2;</code>	<code>//can be written as</code>	<code>x /= 2;</code>
<code>x = x + y;</code>	<code>//can be written as</code>	<code>x += y;</code>
<code>x = x - y;</code>	<code>//can be written as</code>	<code>x -= y;</code>

String Operators



- Operators in general are not defined for String type data
- String concatenation can be performed using
 - +
 - +=
- Only works when at least one operand is a String
- Other operand is converted to a String
- The two Strings are then concatenated into a new String
- Any kind of data can be converted to a String

String Operators Example



String Concatenation

```
String message = "String data ";  
int i = 34;  
float f = 89.13F;  
boolean b = true;  
char c = '*';
```

```
message + i -> "String data 34"  
f + message -> "89.13String data "  
message + b -> "String data true"  
message + c -> "String Data *"  
f + " " + i -> "89.13 34"  
b + " " + c + " " + i -> "true * 34"
```

```
// implicit string conversion  
(i + "") + f -> "3489.13"  
i + f -> 123.13
```


Operator Precedence and Associativity



Operator	Associates
[] . () function_call	Left to right
! ~ ++ -- cast new - +(unary form)	Right to left
* / %	Left to right
+ - (binary form)	Left to right
<< >> >>>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
? :	Left to right
= (op=)	Right to left

Operator Precedence and Associativity

Example



Operator Precedence

Since `*` has a higher precedence than `+`

```
2 * 4 + 3 -> 8 + 3 -> 11
3 + 2 * 4 -> 3 + 8 -> 11
```

But we can change the order of operations with `()` to make the `+` be evaluated first

```
2 * (4 + 3) -> 2 * 7 -> 14
(3 + 2) * 4 -> 5 * 4 -> 20
```

The `&&` operator associates from left to right.
In the following assume `x` is 3, and `y` is 5

```
(x == 3) && (y == 5) && (x == y) -> true && (x == y) -> false
```

which we can override with `()`

```
(x == 3) && ((y == 5) && (x == y)) -> (x == y) && false -> false
```

On the other hand assignment associates from right to left. Assume `y` is 5

```
x = y = 4 -> x = 4 -> 4
```

Widening Conversions



- ◎ Java will always do a widening conversion.
- ◎ When converting a numeric data type to a wider version of the same type, the numeric value is preserved exactly without any loss in precision.
- ◎ Converting any integral data type to a floating point type is allowed. There is no loss of magnitude, but there can be a loss of precision.

Widening Conversions Example I



Widening Conversions

```
long longVar;  
int  intVar;  
short shortVar;  
byte  byteVar;  
char  charVar;  
float floatVar;  
double doubleVar;  
  
byteVar = 120;  
shortVar = byteVar;  
intVar = shortVar;  
longVar = intVar;  
System.out.println("LongVar is "+ longVar); // value is 120L
```

Widening Conversions Example II



Widening Conversions -- integral to floating point

```
long longVar;  
float floatVar;  
double doubleVar;  
  
longVar = Long.MAX_VALUE;  
floatVar = longVar;  
doubleVar = longVar;  
System.out.println("longVar is "+ longVar);  
System.out.println("floatVar is "+ floatVar);  
System.out.println("doubleVar is "+ doubleVar);
```

```
// Output is  
longVar is 9223372036854775807  
floatVar is 9.223372E18  
doubleVar is 9.223372036854776E18
```

Narrowing Conversions and Casting



- Narrowing conversions are the opposite of widening conversions
 - Converting a data type to a smaller version of the same type
 - Converting from a floating point data type to an integral data type
- Java does not perform narrowing conversions automatically
- The data must be cast to the narrower type
 - A cast operator is represented as a new data type name in parentheses placed before the variable or expression to be cast
 - `variable2 = (new_type) variable;`

Narrowing Conversions and Casting Example



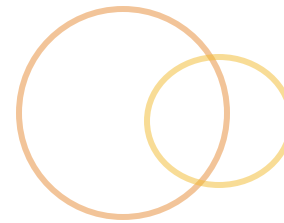
Narrowing conversions

```
byte byteVar = (byte)255;
short shortVar = (short)214748360;
int intVar = (int) 1e20F;
int intVar2 =(int)Float.NaN;
float floatVar = (float)-1e300;
float floatVar2 = (float)1e-100;
```

```
System.out.println("(byte)255 -> " + byteVar);
System.out.println("(short)214748360-> "+ shortVar);
System.out.println("(int)1e20f -> " + intVar);
System.out.println("(int)NaN -> " + intVar2);
System.out.println("(float)-1e300 -> " + floatVar);
System.out.println("(float)1e-100 -> " + floatVar2);
```

```
// produces the output
(byte)255 -> -1
(short)214748360-> -13112
(int)1e20f -> 2147483647
(int)NaN -> 0
(float)-1e300 -> -Infinity
(float)1e-100 -> 0.0
```

String Conversions



- ◎ Any primitive data type can be converted to a `String`.
 - ◎ By implicit `String` conversion
 - ◎ For each primitive data type, we can use a `toString()` function found in “wrapper” classes to convert the value to a `String`
- ◎ We can also convert from a `String` to various data types.
 - ◎ This is more complicated because not every string of characters can be converted to a particular primitive data type.

String Conversions Example



Converting to and from strings

```
String s = "123";  
String t;
```

```
int k = 123;  
float f = 19.801F;
```

```
//Integer and Float are “wrapper” classes  
t = Integer.toString(k);  
t = Float.toString(f);
```

```
// This line will not compile, you can't convert from a  
// String this way  
k = (int)s;
```

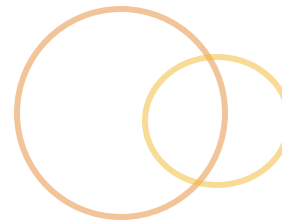
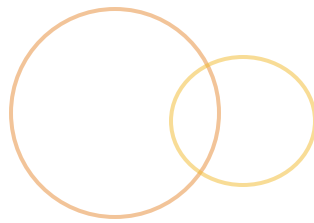
Forbidden Conversions



According to the Java language specification:

- ⦿ No conversion from any reference type to any primitive type
- ⦿ Except for the `String` conversions, no permitted conversion from any primitive type to any reference types
- ⦿ No permitted conversion to `boolean` types
- ⦿ No permitted conversion from `boolean` other to a `String` conversion

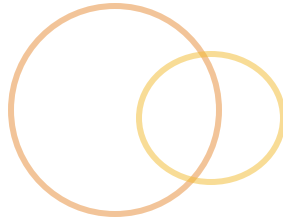
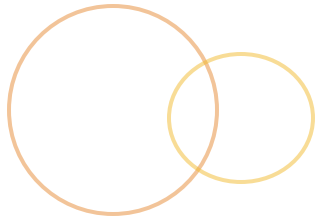
Summary



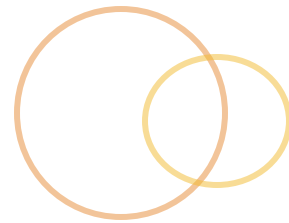
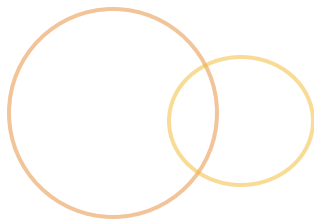
We covered:

- ⦿ What operators and expressions are and how operators are used to create expressions
- ⦿ The operators in Java, the kinds of data they operate on, and what types of expressions they produce
- ⦿ The difference between narrowing and widening operators
- ⦿ Using the cast operator correctly

Static Methods and Fields



Objectives



At the end of this module, you should be able to understand:

- Class loaders
- Static member order of execution
- Static finals
- Class methods and variables
- Static initialization blocks
- Static imports

Java Virtual Machine Class Loader



- ⦿ Java has a special portion of memory to hold global variables.
- ⦿ Java uses memory called the Data Segment to create static members
- ⦿ The Data Segment has two distinct areas:
 - ⦿ Initialized Data
 - ⦿ Stores static variables that have been initialized by program
 - ⦿ Uninitialized Data
 - ⦿ Stores static variables that have not been initialized by program
 - ⦿ Initializes all uninitialized static variables to 0

Primordial Class loader



- ⦿ First step taken by JVM is to load the Primordial Class Loader
- ⦿ Class loader is responsible for converting named classes into the *computer understandable* bits that represent that class.
- ⦿ When we first start a Java program we don't have any classes to create an object.
- ⦿ The JVM must load a class first and call a method that is associated with a class and not an object.
- ⦿ In Java, the egg comes before the chicken.

The Class Loader has Four Responsibilities



- ⦿ Loading a class

- ⦿ A class is loaded when the class loader resolves the class, reads the .class file, converts the .class file into bits, and loads it into memory.

- ⦿ Verifying the class

- ⦿ Class loaders must verify that a class being loaded conforms to a very strict set of parameters.

- ⦿ Preparing the class

- ⦿ A class loader prepares a class by allocating memory that will hold all of the class variables and methods.

- ⦿ Initializing the class

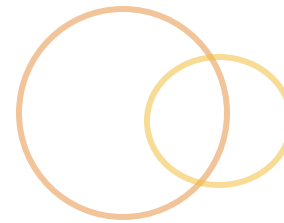
- ⦿ Each class has an `clinit()` method that may be used to set up up a class whenever it is loaded into memory.

The Static Keyword Has Four Options



- ◎ static variables
- ◎ static methods
- ◎ static blocks of code
- ◎ static inner classes

Public Static Finals



- ⦿ Constant class member
- ⦿ `public static final` is used to create a global variable.
- ⦿ `public static final double PI = 3.141592653589793`
- ⦿ `PI` does not change and should be available at all times without having to create an object.
- ⦿ Syntax for global variables is all upper case:

```
public static final double CD_RATE = 2.1;
```

Class Variables and Methods



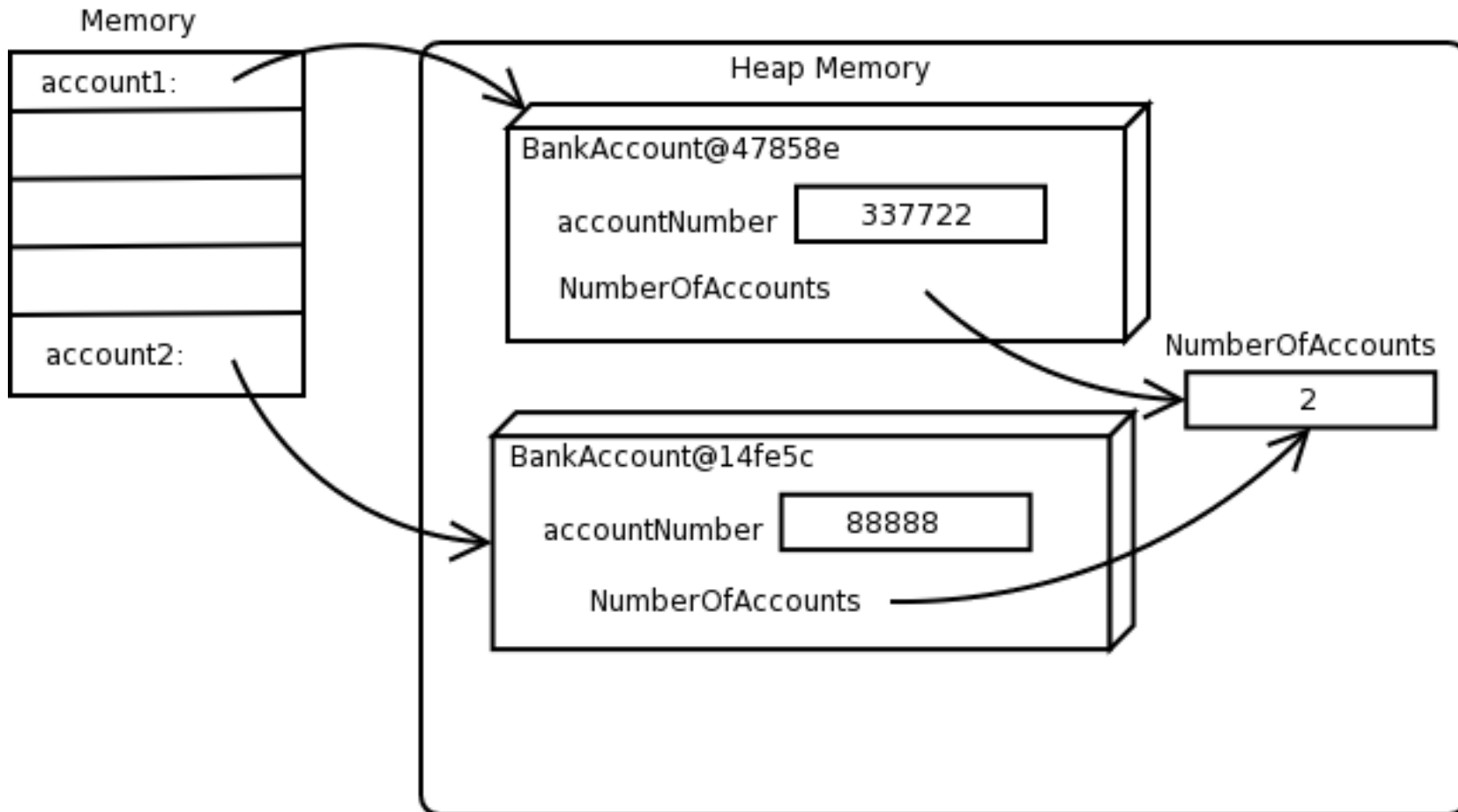
- ◎ Java provides a mechanism to declare variables that belong to the class as a whole not a specific object
 - ◎ Called static or class variables
 - ◎ Provide much of the functionality that was provided by global variables in structured languages
 - ◎ static variables belong to the class and are shared by all instances of the class
- ◎ Think of class or static variables as being variables that are global within a class.
- ◎ Static variables can be accessed by either instance methods or static methods.
- ◎ Use the dot notation to access static variables:
`class_name.staticVariableName`
`BankAccount.NumberOfAccounts`

Class Variables

- To make an instance variable static, you simply place the keyword **static** before the variable definition
- For example, the following defines a static data member and initializes it:

```
class BankAccount {  
    // Class Variables  
    static int NumberOfAccounts = 0;  
  
    // Instance Variables  
    String accountNumber = null;  
    char accountType;  
    private float balance;  
    int accountStatus;  
    // rest of class definition  
}
```

Referencing Static Variables



Referencing Static Variables



```
class BankAccount {  
    // Class Variables  
    static int NumberOfAccounts = 0;  
    // Instance Variables  
    String accountNumber = null;  
    char accountType;  
    private float balance;  
    int accountStatus;  
  
    // Using static variable in a constructor  
    BankAccount(String num, char type, float bal) {  
        accountNumber = num;  
        accountType = type;  
        balance = bal;  
        accountStatus = (type == 's')? 100: 0;  
        NumberOfAccounts++;  
    }  
}
```

Static Variable Initialization



- There are three mechanisms in Java to initialize static variables:
 - Default initialization – exactly like default instance variable initialization
 - Explicit initialization – exactly like explicit instance variable initialization
 - Static Initializer* – similar to an initializer
- A static initializer
 - Initializes the static member variables of the class
 - Allows complex computations, such as loops and conditions
- A block of code, identified with the keyword `static`, is allowed in the class definition
 - This block of code is executed when the static variables actually come into existence
 - The `static` block is intended to be used to initialize the static variables, nothing more
 - A static initializer is executed last, after default and explicit initialization of the class

Static Initialization Block



```
class BankAccount {  
    // Class Variables  
    static int NumberOfAccounts;  
    // Instance Variables  
    String accountNumber = null;  
    char accountType;  
    private float balance;  
    int accountStatus;  
  
    static { //static initializer  
        NumberOfAccounts = 0;  
    }  
  
    // Using static variable in a constructor  
    BankAccount(String num, char type, float bal) {  
        accountNumber = num;  
        accountType = type;  
        balance = bal;  
        accountStatus = (type == 's')? 100: 0;  
        BankAccount.NumberOfAccounts++;  
    }  
}
```


Static Methods



- ◎ Java allows methods that are associated with the class
 - ◎ They are not associated with any specific object.
 - ◎ You can access the methods without an object.
 - ◎ All static methods exist independently of any objects.
- ◎ Static methods are typically used for:
 - ◎ Library functionality (`Math.abs()`, `Integer.parseInt()`)
 - ◎ Implementing certain design patterns (*Factory*, *Singleton*, etc)

Static Methods



- ◉ There are some rules when dealing with static methods
 - ◉ There is no “current object” in a static method, therefore you cannot directly reference instance variables and methods inside a static method
 - ◉ Conversely, however, instance methods can access static variables and static methods
 - ◉ Static methods can access instance variables if a reference to an instance is used
- ◉ Static methods are accessed using the dot notation

Static Method Example



```
class BankAccount {  
    // Class Variables  
    private static int NumberOfAccounts;  
    // Instance Variables  
    String accountNumber = null;  
    char accountType;  
    private float balance;  
    int accountStatus;  
    static {  
        NumberOfAccounts = 0;  
    }  
    // Class Methods  
    private static void incrementCount() {  
        BankAccount.NumberOfAccounts++;  
    }  
    public static int numActs() {  
        return BankAccount.NumberOfAccounts;  
    }  
}
```

. . .

Static Method Example (cont.)



```
// Using static variable in a method
BankAccount(String num, char type, float bal) {
    accountNumber = num;
    accountType = type;
    balance = bal;
    accountStatus = (type == 's')? 100: 0;
    incrementCount();
}
}

class Test{
    public static void main(String [] args) {
        BankAccount b = new BankAccount();
        System.out.println("Number of Accounts: "+
b.numActs());
        System.out.println("Number of Accounts: "+
BankAccount.numActs());
    }
}
```

Static Access to Instance Example

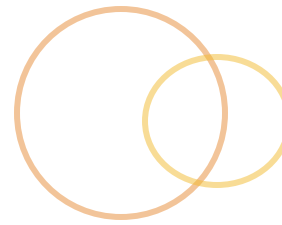


```
class ThingTwo {  
    int x; // instance variable  
    static void whatsX() {  
        // This would fail, doesn't know what x we mean:  
        System.out.println("x is " + x);  
        // This works fine:  
        ThingTwo aThing = new ThingTwo();  
        System.out.println("x is " + aThing.x);  
    }  
}
```

Static Imports

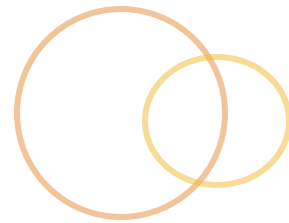
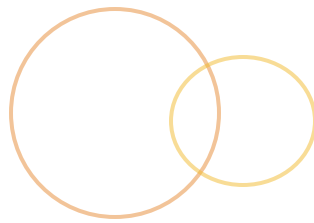
- Java 1.5 introduced static imports.
- Static imports are a shortcut for accessing a variable without a class name
- When using a static import statement, we can avoid having to use the class name to access static variables and methods.

Static Inner Classes



- The **static** keyword can be used when declaring a static inner class, sometimes called a nested class.
- Inner classes are classes that exist within another class.
- Static inner classes could be used to provide a common property for each instance of the class.
- An example use for a static inner class may be to represent a common property like the size of a brush in a Draw program.

Summary



We covered

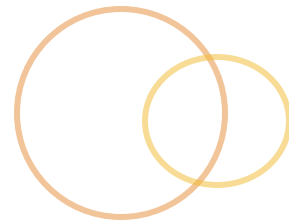
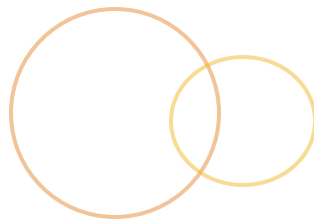
- Class loaders
- Static member order of execution
- Static finals
- Class methods and variables
- Static initialization blocks
- Static imports

Control Structures in Java

Part I



Objectives



At the end of this section, you should be able to:

- ② Describe statements and blocks
- ② Describe local variables and their scope
- ② Describe the flow of control of a Java program
- ② Use **if** and **switch** statements

Statements and Blocks



- Program execution is defined by statements
 - Statements can extend over any number of lines.
 - Statements are terminated by a semi-colon (;).
 - Executable statements only exist inside blocks (usually, but not exclusively, methods).
- Statements can include expressions
 - An expressions has both a compile time and a runtime type, and also has a runtime value.
 - Statements do not have to have these (but may).
 - The **while** loop is a statement that is not an expression.
- An expression can be used as a statement
 - e.g.: `x = 3;`

Syntax of Statements & Expressions



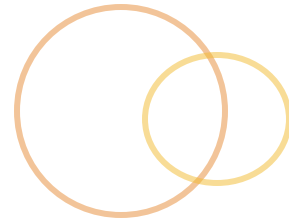
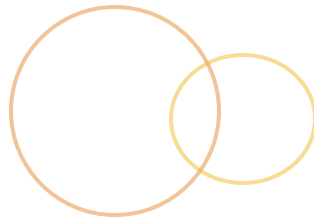
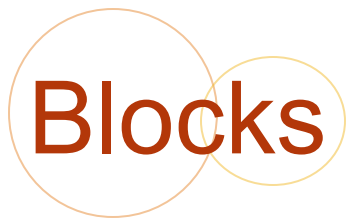
```
x = 7 + 1      // expression - not a statement (incomplete)
x = 7 + 1;     // adding a semi-colon makes this a statement.
int x;         // statement - not an expression
x++;          // expression and a statement
```

```
// a statement and complex expression
int y = x++ / ( z * 2.0 );
```

```
// the empty statement - legal but pointless
;
```

```
// one line containing three statements
x = 7 + 1; x++; System.out.println(x);
```

```
// one statement spread over three lines
x =
    7 +
    1;
```



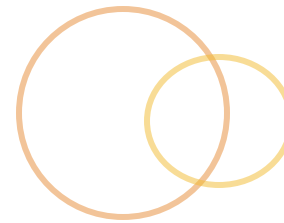
- A block is a sequence of statements enclosed in braces { }
- A block can occur anywhere in a Java program that a statement can
 - Essentially, a block *is* a statement
- Any statement in the block could itself be replaced by a block
- Nested blocks are allowed

Blocks and Nested Blocks Example



```
public class BlocksAndStatements {
    public static void main(String []args) {
        int x = 0;
        x++;
        int y = 0;
        { // Start of a user defined block
            { // Start of a nested user defined block
                System.out.println(x);
                System.out.println(y);
            } // End of a nested user defined block
            y = x % 3;
        } // End of the outer user defined block
        y++;
        { // Start of a second user defined block
            System.out.println(x);
            System.out.println(y);
        } // End of the second user defined block
    } // End of the block that makes up the body of the main method
} // end of the block that makes up the class definition
```

Local Variables



- Local variables exist within a method and have the scope of the enclosing block
 - Are **not** automatically initialized
 - Used as temporary or “working” variables within the body of a block
- Local variables’ lifetime is *generally* controlled by program flow
 - Come into existence when the flow of control passes through their declaration
 - Cease to exist when the flow of control passes out the defining block

Scope of a Local Variable

Example I



```
// scope of outerVar
void someMethod(int x) {
    System.out.println("Entering someMethod..");
    int outerVar = 1; //local to someMethod
    { //"inner block"
        System.out.println("Entering inner block...");
        int innerVar = 4; //local to "inner block"
        innerVar = outerVar + x;
        System.out.println("innerVar is "+ innerVar);
    }
    System.out.println("outerVar is "+ outerVar);
}
```


Scope of a Local Variable

Example II



```
// scope of innerVar
void someMethod(int x) {
    System.out.println("Entering someMethod..");
    int outerVar = 1; //local to someMethod
    { //"inner block"
        System.out.println("Entering inner block...");
        int innerVar = 4; //local to "inner block"
        innerVar = outerVar + x;
        System.out.println("innerVar is "+ innerVar);
    }
    // innerVar does not exist here...
    // System.out.println("innerVar is "+ innerVar); would fail!
    System.out.println("outerVar is "+ outerVar);
}
```

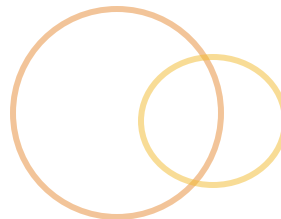
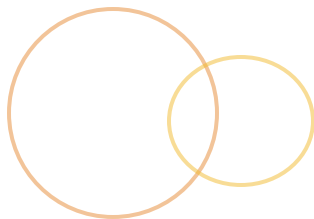
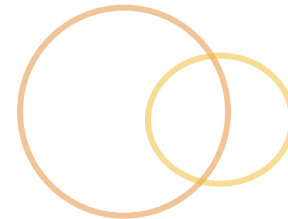
Common Local Variable Errors

Example



```
// scope of innerVar
void someMethod(int x) {
    System.out.println("Entering someMethod..");
    int outerVar = 1;
    {
        System.out.println("Entering inner block...");
        // Error 1: referencing innerVar before it is declared
        System.out.println(innerVar);
        int innerVar = 4;
        innerVar = outerVar + x;
        System.out.println("innerVar is "+ innerVar);
        // Error 2: trying to declare a variable with a name
        // used by another local variable in the same scope
        int outerVar = 10;
    }
    System.out.println("outerVar is "+ outerVar);
    // Error 3: Trying to reference innerVar outside of its scope
    System.out.println(innerVar);
}
```

Flow Control



Basic `if` Statement Syntax



- ◎ The `if` statement looks like:

```
if (<test-expression>)  
    <subordinate-statement>
```

- ◎ The expression must be contained in parentheses.
- ◎ The test-expression must be of Boolean type.
- ◎ The subordinate statement of `if` should usually be a block.
- ◎ If the subordinate statement is a single statement, it must be terminated by a semicolon.

Basic if Statement Syntax Example



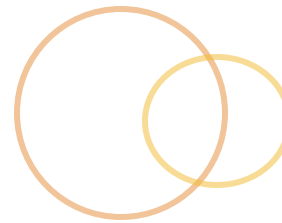
```
// if statement with a single statement as a then-clause  
// !!!Avoid this style!!!
```

```
if (x == 3)  
    System.out.println("x is, in fact, three");  
x = 24;
```

```
// if statement with a body as a then-clause
```

```
boolean test = (x > 23);  
if (test) {  
    System.out.println("x is out of range");  
    x = x - 10;  
    System.out.println("value of x is reset to" + x);  
}  
x = 24;
```

if-else Statements



- ⦿ The **if-else** form allows two mutually exclusive paths of execution
- ⦿ The **if-else** form of the **if** statement looks like

```
if (<test-expression>)  
    <subordinate-statement-1>  
else  
    <subordinate-statement-2>
```

- ⦿ If the test expression is `true`, the **then** clause executes exactly as we just saw in the previous section
 - ⦿ If the test condition is `false`, the **else** clause executes instead
 - ⦿ Since `test-expression` is Boolean, one of the clauses will always execute.

if-else Statements Example



```
// if statement with a single statements
// in both then and else clauses. !!! Avoid this style !!!
if (x == 3)
    System.out.println("x is, in fact, three");
else
    System.out.println("x is NOT, in fact, three");

// if statement with a body in both then and else clauses
if (x > 23) {
    System.out.println("x is out of range");
    x = x - 10;
    System.out.println("value of x is reset to" + x);
}
else {
    System.out.println("x is in range");
    x++;
}
```

if-else Statements Example (cont.)



```
// if statements with one block-body and one  
// statement body !!! Avoid this style !!!  
if (x > 23) {  
    System.out.println("x is out of range");  
    x = x - 10;  
    System.out.println("value of x is reset to" + x);  
}  
else  
    System.out.println("x is in range");  
  
if (test)  
    System.out.println("x is out of range");  
else {  
    System.out.println("x is in range");  
    x++;  
}
```


The Dangling else Problem



```
if (test1)
    if (test2)
        System.out.println("test1 and test2 true");
else
    System.out.println("test1 if false");
```

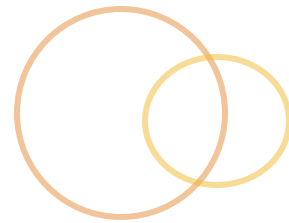
// what the programmer meant was

```
if (test1) {
    if (test2)
        System.out.println("test1 and test2 true");
}
else
    System.out.println("test1 if false");
```

// what the compiler saw was

```
if (test1) {
    if (test2)
        System.out.println("test1 and test2 true");
    else
        System.out.println("test1 if false");
}
```

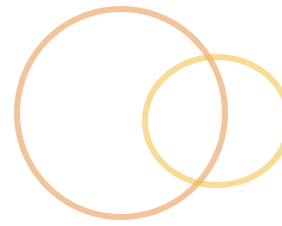
if-else Statements



- Most programming languages provide an alternate form to nested if statements
- In Java the syntax is simply a sequence of if/else constructs

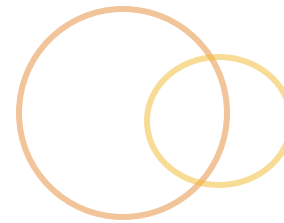
```
if (test1) first-then-clause  
else if (test2) second-then-clause  
else if (test3) third-then-clause  
else else-clause /* the else clause is optional */
```

if-else Statements



```
int status = getStatus();
if (status == 0) {
    /* stuff to do if status is 0 */
}
else {
    if (status == 1) {
        /* stuff to do if status is 1 */
    }
    else {
        if (status == 2) {
            /* stuff to do if status is 2 */
        }
        else {
            /* stuff to do if status is anything else */
        }
    }
}
```

if-else Example



```
int status = getStatus();  
if (status == 0) {  
    /* stuff to do if status is 0 */  
} else if (status == 1) {  
    /* stuff to do if status is 1 */  
} else if (status == 2) {  
    /* stuff to do if status is 2 */  
} else {  
    /* stuff to do if status is anything else */  
}
```

The **switch** Statement



- ⦿ Sometimes, it is necessary to perform conditional behavior based on differing numeric values

```
if(x == 2) //do something
else if(x == 3) // do something else
else if(x == 4) //do something else
else //do something default
```

- ⦿ Using **if**, **else if**, and **else** can be
 - ⦿ Tedious (especially if you have many conditions)
 - ⦿ Cause unwanted overhead (each condition is evaluated until the right one is found)
- ⦿ There is a control construct that helps with this – **switch**
 - ⦿ **switch** only permits `int`, `char`, and `enum` values

The switch Statement



```
// testvar is a variable of some type
switch(testvar) {
    case value1:
        /* code to execute when testvar has the value value1 */
        break;
    case value2:
        /* code to execute when testvar has the value value2 */
        break;
    case value3:
        /* code to execute when testvar has the value value3 */
        break;

    /*--- more cases ---*/
    case value_n:
        /* code to execute when testvar has the value value_n */
        break;
    default:
        /* code to execute when testvar none of the above values */
        break;
}
```

The switch Statement Example



```
char status = 'a';
```

```
switch (status) {  
    case '*' :  
        System.out.println("Asterisk");  
        break;  
    case 'a' :  
        System.out.println("letter a");  
        break;  
    case 'z' :  
        System.out.println("letter z");  
        break;  
    default :  
        System.out.println("Unrecognized character");  
        break;  
}
```

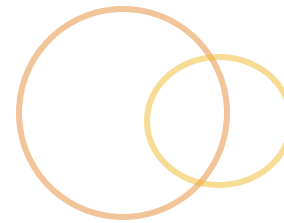
The switch Statement (cont.)



```
System.out.println("And continuing...");
```

```
status = 'g';
switch (status) {
    case '*':
        System.out.println("Asterisk");
        break;
    case 'a':
        System.out.println("letter a");
        break;
    case 'z':
        System.out.println("letter z");
        break;
    default:
        System.out.println("Unrecognized character");
        break;
}
System.out.println("And continuing...");
```


The case Statement



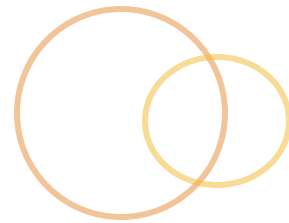
- Ordering of **case** statements is entirely up to the programmer
 - The `default` case does not have to be the last **case** statement
 - The order of **case** statements does not have to follow the logical ordering of integers
 - The `default` case is optional
- Execution will continue into the next **case** statement if a **break** statement is not encountered
 - This called *fall through*
 - Fall-through behavior allows the same code to be applied to a set of test cases
- Applying fall-through behavior can achieve some advanced solutions but is prone to being misread. Most missing breaks are accidental omissions, take care with this!

switch Statement Fall-through Example



```
char status = 'z';
switch (status) {
    case '*': // does nothing, but then falls through to 'a'
    case 'a':
        System.out.println("letter a or an asterisk");
        break;
    case 'z':
        System.out.println("letter z");
    case 'w':
        System.print("letter w");
        break;
    default:
        System.out.println("Unrecognized character");
        break;
}
```

The Ternary Operator



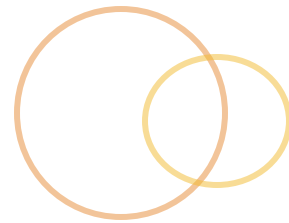
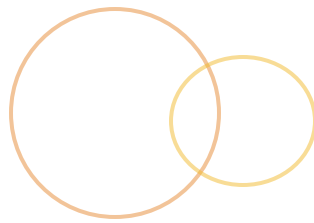
- There is one ternary operator in Java
 - Inherited from C and C++
 - The same as an **if-else** statement but is an expression
- The conditional operator is `? :`
`test-expression ? then-expression : else-expression`
- The test-expression must be a Boolean expression
 - If `test-expression` evaluates to `true`, `then-expression` is evaluated and the result returned.
 - If `test-expression` is `false`, `then else-expression` is evaluated and returned.

The Ternary Operator Example



```
class Ternary {  
    public static void main(String [] args) {  
        String message;  
        boolean test = false;  
        // Using an if statement  
        if (test){  
            message= "Then clause";  
        }  
        else {  
            message = "Else clause";  
        }  
        System.out.println(message);  
        // Using a conditional operator  
        System.out.println(test ? "Then expression" :  
                                "Else expression");  
        System.out.println(!test ? "Then expression" :  
                                "Else expression");  
    }  
}
```

Summary



In this section we covered

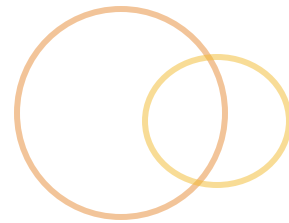
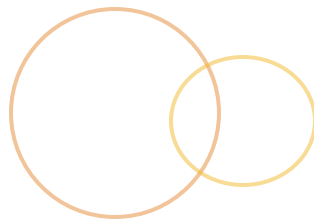
- Statements and blocks
- Local variables and their scope
- The flow of control in Java programs
- **if** and **switch** statements

Control Structures in Java

Part II



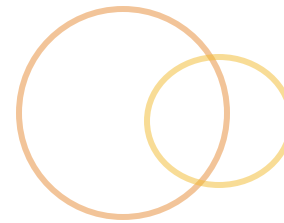
Objectives



At the end of this section, you should be able to:

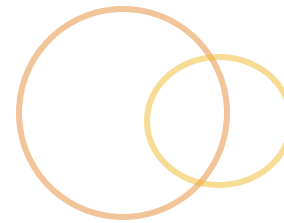
- Use **while**, **do-while** and **for** loops
- Use **break** and **continue** statements
- Use method overloading correctly

Loops in Java



- There are three looping structures in Java
 - while** loops
 - do-while** loops
 - for** loops
- The **while** and **do-while** loops are more natural for iterating until a condition occurs.
- for** is more natural for “known distances,” for keeping an index available, and for working with arrays and other collections.

while Loops



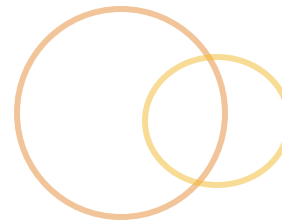
- The most basic looping structure
 - Condition is evaluated prior to executing the body
 - The body is executed as long as a condition is true

- The **while** loop looks like this:

```
while (test-condition)  
    <body statement>
```

- As with conditions, `<body-statement>` should usually be a block
- `test-condition` must be a Boolean expression

while Example



```
int count = 0;
```

```
// while loop with a statement as a loop body
```

```
// !!! Avoid this style !!!
```

```
while (count < 10)
```

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

```
// while loop with a block as a loop body
```

```
count = 0;
```

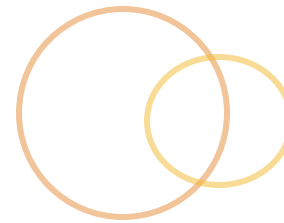
```
while (count < 10) {
```

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

```
    count++;
```

```
}
```

do-while Loops



- ⦿ A variation on the **while** loop is the **do-while** loop
 - ⦿ Always executes the body of the loop at least once
 - ⦿ Determines subsequent execution based on condition

- ⦿ The **do-while** loop looks like this:

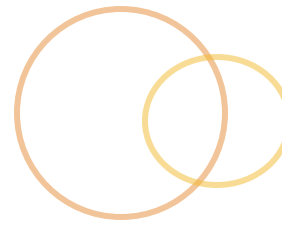
```
do
```

```
<body-statement>
```

```
while (test-condition);
```

- ⦿ `<body-statement>` should usually be a block
- ⦿ The **do** indicates the start of the loop body and the test condition appears after the **while**
 - ⦿ Notice that there is a semicolon after the test condition
 - ⦿ If the loop body is a statement it must be terminated with a semicolon.

do-while Example



```
int count = 0
// do-while loop with a statement as a loop body
// !!! avoid this style !!!
do
    System.out.println("count is "+ count++);
while (count < 10);
// do-while loop with a block as a loop body
count = 0;
do {
    System.out.println("count is "+ count);
    count++;
}while (count < 10);
```

Command-line Input Example I



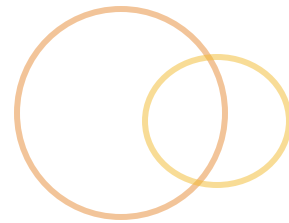
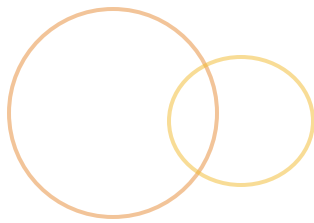
```
class DoWhileTest1 {
    public static void main(String [] args) {
        // Using a while loop
        try {
            char input;
            String output = "";
            input = (char)System.in.read();
            while(input != '\n') {
                output = input + output;
                input = (char)System.in.read();
            }
            System.out.println(output);
        }
        catch (Exception e) {
            System.out.println("IO Exception:" + e);
        }
    }
}
```

Command-line Input Example II



```
class DoWhileTest2 {  
    public static void main(String [] args) {  
        //Using a do-while loop  
        try {  
            char input;  
            String output = "";  
            do {  
                input = (char)System.in.read();  
                output = input + output;  
            } while(input != '\n');  
            System.out.println(output);  
        }  
        catch (Exception e) {  
            System.out.println("IO Exception:" + e);  
        }  
    }  
}
```

for Loops

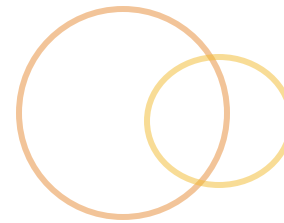


- ② **for** loops are intended to make iterations involving counters or indexes easier to program
- ② They have the following structure

```
for (initial-clause; test-clause; iteration-clause)  
    loop-body
```

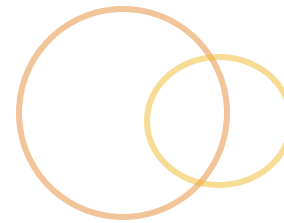
- ② It is not required that you provide a valid expression for each clause, we will see more on this later.

for Example I



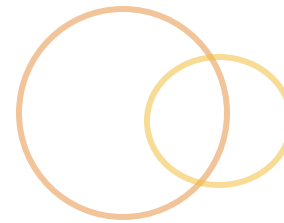
```
class ForLoop {  
    public static void main(String [] args) {  
        int sum = 0;  
        int index;  
        for (index = 1; index <=100; index++) {  
            sum += index;  
        }  
        System.out.println("Sum =" +sum);  
    }  
}
```


for Example II



```
class ForLoop2 {
    public static void main(String [] args) {
        int [] forwards = {0,1,2,3,4,5,6,7,8,9};
        int [] backwards = {0,1,2,3,4,5,6,7,8,9};
        int idx1, idx2;
        for (idx1 = 0, idx2 = 9; idx1 <forwards.length; idx1++, idx2--){
            backwards[idx2]=forwards[idx1];
        }
        System.out.println("Backwards is now..");
        for (idx1 = 0; idx1 <backwards.length; idx1++){
            System.out.println(backwards[idx1]);
        }
    }
}
```

Local Variables



- Remember, local variables are defined within blocks.
- Their existence is defined by their scope.
- Local variables can be used in loops to hold temporary data.
- Local variables can be defined as part of the loop **test** expression or as variables in the body.

Local Variables in for Loops



```
class ForLoop3 {
    public static void main(String [] args) {
        int outer = 0; //local to main
        //middle is defined in the for construct
        for (int middle = 0; middle <10; middle++) {
            int inner = 0; //same scope as middle
            inner++;
            outer++;
            System.out.println("outer="+outer+" middle="+middle+
                               " inner="+inner);
        }
        System.out.println(" outer="+outer);
        // This following two lines will not compile because
        // they are out of scope for middle and inner.
        //System.out.println(" inner="+inner);
        //System.out.println(" middle="+middle);
    }
}
```

Local Variables Example Output



Problems @ Javadoc Declaration Console

<terminated> Arrays [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home/bin/java

```
outer=1 middle=0 inner=1
outer=2 middle=1 inner=1
outer=3 middle=2 inner=1
outer=4 middle=3 inner=1
outer=5 middle=4 inner=1
outer=6 middle=5 inner=1
outer=7 middle=6 inner=1
outer=8 middle=7 inner=1
outer=9 middle=8 inner=1
outer=10 middle=9 inner=1
  outer=10
```

More on Local Variables in Loops



Local variables can sometimes cause surprises

```
// This is okay  
int k; String s;  
for (k=23, s="hi";;  
{ /*body */ }
```

```
// As is this  
for (int k=0, j=1;;)  
{/* body */ }
```

```
// This doesn't work -- compiler thinks j is being redeclared.  
int j;  
for (int k =0, j=1;;)  
{ /* body */ }
```

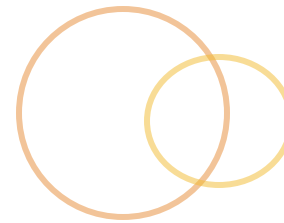
```
// Nor does this -- compiler does not expect to find "String"  
// type must be consistent across all declared variables  
for (int k=0, String s="hi";;) { /* body */ }
```

Variations on the for Loop



```
// Variation one - an infinite for loop
for (;;) {
    // Only way to end this one is with a break
    // This can be used to poll a resources indefinitely
    if (conditions) break;
}
// Variation two - only a test expression
// Equivalent to a while loop
int k = 0;
for (; k < 10;) {
    k++
}
// Variation three - just an initialization clause
int z;
for (int k = 0, z=35;;) {
    if (++j >0) break;
}
// Variation four - just an iteration clause
int z=35;
for (;;z++) {
    if (z >100) break;
}
```

for-each Loop



- ⦿ What is a **for-each** loop?
 - ⦿ A variation of a **for** loop
 - ⦿ Simplified notation targeted at collections and arrays
- ⦿ Why does it exist?
 - ⦿ Removes redundant iteration code
 - ⦿ Simplifies iteration over collections

for-each Loop (cont.)



- ⦿ How does it work?
 - ⦿ Functions like a `for` loop . .
 - ⦿ Iterates over collection
 - ⦿ Accesses each collection element individually
 - ⦿ . . but has a different syntax
 - ⦿ Has an initialization “clause” and “expression” clause
 - ⦿ Initialization clause holds current element in collection
 - ⦿ Expression clause defines collection
 - ⦿ Does not have a “test” clause or “increment” clause
 - ⦿ Clauses separated by a colon instead of a `semicolon`
 - ⦿ Translated into a formal `for` loop at compile time

Simple for-each Loop Example



```
1  package examples.foreach;
2
3  /** ... */
10 public class SimpleForEach {
11
12     public static void main(String[] args) {
13         //initialization clause - String s
14         //expression clause - args
15         for(String s : args) {
16             System.out.println(s);
17         }
18     }
19 }
20
```

Limitations of `for-each` Loop



- ⦿ Limited type support
 - ⦿ Supports arrays
 - ⦿ Supports `java.lang.Iterable`
 - ⦿ Collections support `Iterable` through class hierarchies
 - ⦿ `Iterable` provides `java.util.Iterator` to the **`for-each`** loop
- ⦿ Limited insight
 - ⦿ No way to determine “where am I” during iteration
 - ⦿ No access to iterator or index
- ⦿ Cannot leave parts out
 - ⦿ Would not be meaningful in this loop

Iterable for-each Loop Example



```
1 package examples.foreach;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 /**
7  * The following illustrates using the for-each
8  * loop with a collection through the Iterable
9  * interface.
10 */
11 public class IterableForEachExample {
12
13     public static void main(String[] args) {
14         //convert the array into a list
15         List argList = Arrays.asList(args);
16
17         //iterate over the list
18         for(Object arg : argList) {
19             System.out.println(arg);
20         }
21     }
22 }
23
```

The **break** Statement



- ⦿ Flow control constructs preceding a **break** statement executed until some condition fails.
- ⦿ In some cases, it is necessary to stop the execution of the loop structure due to some “local” condition.
- ⦿ The **break** statement in Java produces an abrupt termination of control.
- ⦿ As soon as the **break** statement is encountered, the processing breaks out of the loop and goes to the first statement after the loop body.
- ⦿ It is possible to utilize nested breaks within nested loops.

The **break** Statement

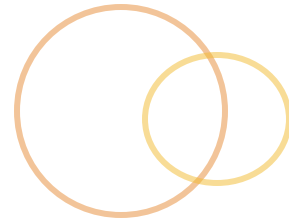
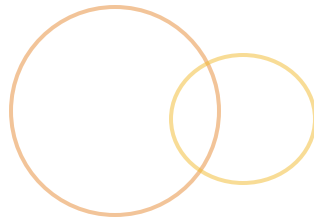


```
class BreakTest {
    public static void main(String [] args) {
        int [] values = {9,45,1,0,98,102,-34};
        int idx = 0;
        while (idx < values.length) {
            if (values[idx] == 98)
                break; //break out of while
            idx++;
        }
        if (idx == values.length) {
            System.out.println("98 was not found");
        }
        else {
            System.out.println("98 found at index "+ idx +" in values");
        }
    }
}
```

Nested Loop break Example



```
int [] target = {9, 45, 1, 0, 98, 102, -34};
int [] test = { 9, 30, 102, 14 };
int idx1 = 0;  int found = -1;
// Outer loop
while (idx1 < test.length) {
    int idx2 = 0;
    while (idx2 < target.length) { // Inner loop
        if (test[idx1] == target[idx2]) {
            found = idx2;
            break; // break inner loop
        }
        idx2++;
    }
    idx1++;
}
if (found == -1) { System.out.println("No test values found"); }
else {
    System.out.println("Found test value "+ target[found] +
        " at index "+ found +" in target");
}
```



- ⦿ When dealing with nested loops, using `break` may not provide the level of termination precision required
- ⦿ For example, you may want to break out of the entire looping structure when something fatal occurs
- ⦿ Java provides a supporting construct called *labels*
- ⦿ Anything in Java can be labeled; however labels really only make sense in the context of loops
- ⦿ The basic syntax of a label is
`label_name: statement`
- ⦿ Labels used with `break` tell the JVM specifically where to terminate

The **break** Statement



```
int [] target = {9,45,1,0,98,102,-34};
int [] test = { 9, 30, 102, 14 };
int idx1 = 0;
int found = -1;
zippy: while (idx1 < test.length) {// Outer loop labeled zippy
    int idx2 = 0;
    while (idx2 < target.length) { // Inner loop
        if (test[idx1] == target[idx2]) {
            found = idx2;
            break zippy; //stop the execution of zippy:while
        }
        idx2++;
    }
    idx1++;
}
if (found == -1) {System.out.println("No test values found");}
else {
    System.out.println("Found test value "+ target[found] +
        " at index "+ found +" in target");
}
```


The `continue` Statement



- In some cases, breaking out of a loop may not be desired.
- Instead of breaking out of the loop, using the **`continue`** statement the current iteration is cancelled and the next iteration is started.
- Just like `break` statements, `continue` statements can be labeled or unlabeled.

continue Statement Example I



- If the remainder after division by 2 (the modulus operator) is not 0, then we have an odd number so we just start the next iteration and skip over the output statement.

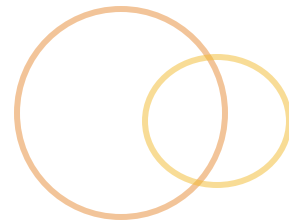
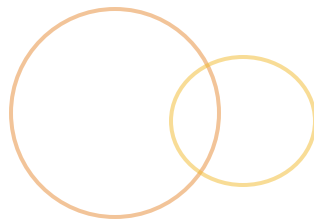
```
int [] target = {9,45,1,0,98,102,-34};
for (int idx = 0; idx < target.length; idx++) {
    if (target[idx] %2 != 0) {
        continue;
    }
    System.out.println(target[idx]+" is even");
}
```

continue Statement Example II



```
class ContinueTest {
    public static void main(String [] args) {
        int [] target = {9,45,1,0,98,102,-34};
        int [] test = { 9, 30, 102, 14 };
        // Outer loop with label zippy
        zippy: for (int idx1= 0; idx1 < test.length; idx1++) {
            for (int idx2 = 0; idx2 < target.length; idx2++) {
                if (test[idx1] == target[idx2]) {
                    System.out.println("Found "+ test[idx1] +
                                         " at " + idx2);
                    continue zippy;
                }
            }
        }
    }
}
```

Summary



We covered

- ◉ Statements and blocks
- ◉ Described local variables and their scope
- ◉ Flow of control of a Java program
- ◉ **if** and **switch** statements
- ◉ **while**, **do-while**, **for**, and **for-each** loops
- ◉ **break** and **continue** statements