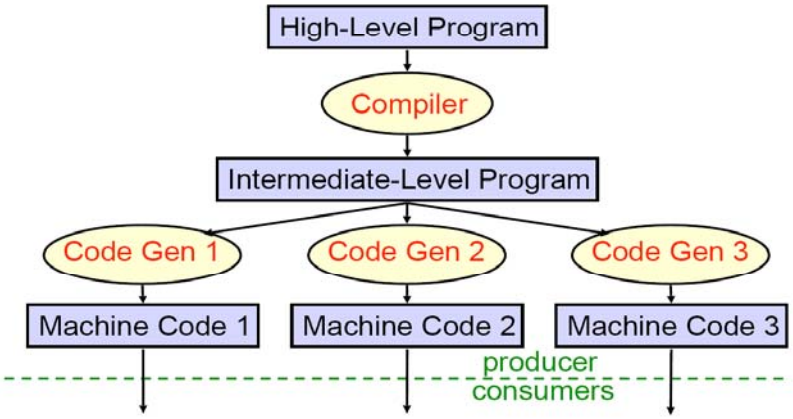
JAVA Language Review

The Basics

Introduction:
Compiling for Different Platforms

- Program written in some high-level language (C, Fortran, Pascal, ...)
- Compiled to intermediate form
- Optimized
- Code generated for various platforms (machine architecture + operating system)
- Consumers download code for their platform

Introduction:
Compiling for Different Platforms



Introduction:
Problem: Too Many Platforms!

- Operating systems:
  - DOS, Win95, 98, NT, ME, 2K, XP, Vista, ...
  - Unix, Linux, FreeBSD, Aix, HP-UX, ...
  - VM/CMS, OS/2, Solaris, Mac OS X, ...
- Architectures:
  - Pentium, PowerPC, Alpha, SPARC, MIPS, ...

Introduction:
Dream: Platform Independence

- Compiler produces *one* low-level program for all platforms
- Executed on a *virtual machine* (VM)
- A different VM implementation needed for each platform, but installed only once

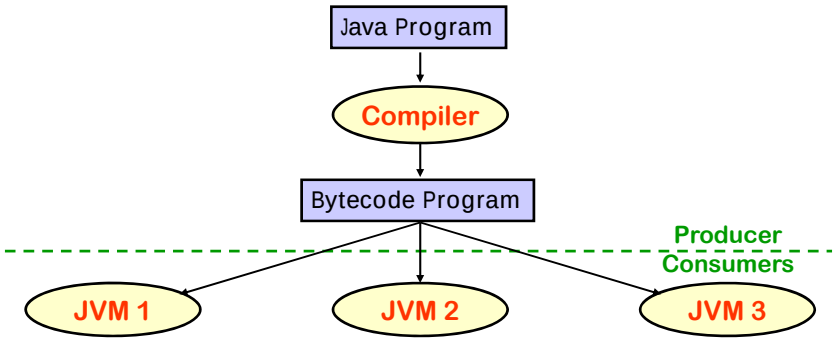
Introduction:
Java Language Features

- Latest, popular, interpreted object-oriented programming language.
- Java comes from the same family of the languages as C++, Pascal, or Basic, adopting ideas from other languages like Smalltalk and Lisp.
- Easy to learn. The designer of C++ once wrote, "*Within C++, there is a much smaller and cleaner language struggling to get out*," many think that the name of that smaller and cleaner language is **Java**

Introduction:
Java Language Features

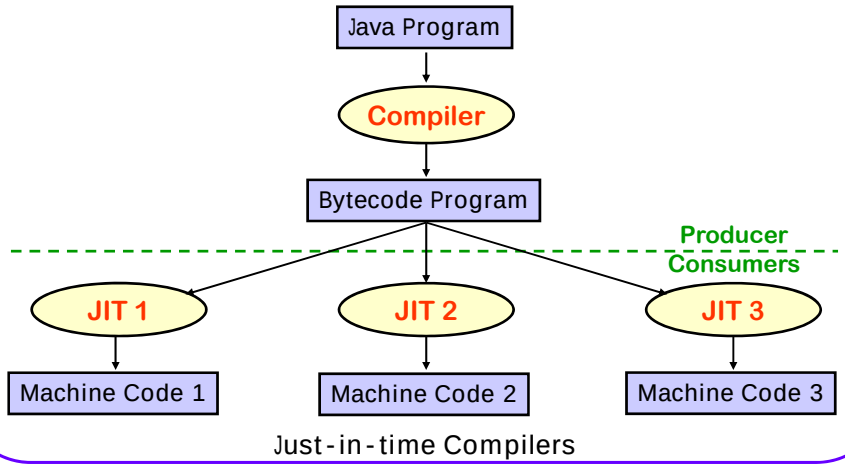
- Strongly typed to minimize programming errors
- Provides:
  - Strong security; Untrusted code cannot do any harm
  - A well-designed GUI support
  - Easy database access
  - Libraries for network communication, and for encryption
- Creates applications that are truly portable (independent of all hardware & operating systems)

Introduction:
Java Platform Independence (1)



Java Virtual Machines

Introduction:
Java Platform Independence (2)



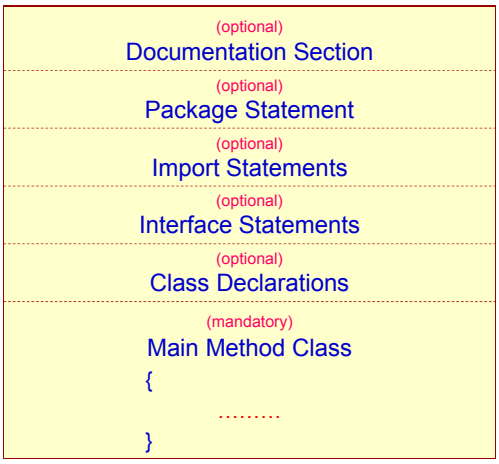
Introduction:
Java Program Structure

- A Java program consists of one or more *compilation units* of Java source code (.java files)
- Each *compilation unit* begins with an optional *package* declaration to specify the *namespace* within which it will define new names
- Followed by zero or more *import* declarations to specify *namespaces* from which it will use other already defined names
- Followed by zero or more *reference type*: *class*, *interface*, *enum* or *annotation* definitions

Introduction:
Java Program Structure

- The definitions include *members* such as *fields*, *methods*, and *constructors*
- *Methods* are *blocks* of Java code comprised of *statements*
- Most *statements* include *expressions*
- *Expressions* are built using *operators* and *values* known as *primitive data types*
- *Statements* are written using *keywords*
- *Keywords*, characters that represent *operators* and *literal* values are all ***tokens***.

Introduction:
Java Program Structure



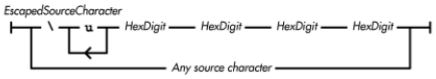
Introduction:
Example Java Program

```
/**
 * This program computes the factorial of a number
 */
public class Factorial {                        // Define the "main method" class
  public static void main(String[] args) {      // The program starts here
    int input = Integer.parseInt(args[0]);      // Get the user's input
    double result = factorial(input);           // Compute the factorial
    System.out.println(result);                 // Print out the result
  }                                             // The main() method ends here

  public static double factorial(int x) {       // This method computes x!
    if (x < 0)                                  // Check for bad input
      return 0.0;                               // If bad, return 0
    double fact = 1.0;                          // Begin with an initial value
    while(x > 1) {                              // Loop until x equals 1
      fact = fact * x;                          // Multiply by x each time
      x = x - 1;                                // And then decrement x
    }                                           // Jump back to start of loop
    return fact;                                // Return the result
  }                                             // The factorial() method ends here
}                                               // The Factorial class ends here
```

Language Basic Components
The Unicode Character Set

- Java programs are written using 16-bit Unicode characters
- Written to file using UTF-8 encoding, which converts the 16-bit characters into a stream of bytes
- A Unicode character can be written using the Unicode escape sequence
- Format: \u*xxxx*

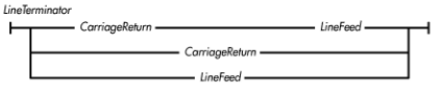


  - Where *x* is a *HexDigit*
  - A *HexDigit* is either a *Digit* or one of the following letters: **A**, **a**, **B**, **b**, **C**, **c**, **D**, **d**, **E**, **e**, **F**, or **f**.
  - A *Digit* is one of the following characters: **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, or **9**.

Language Basic Components
The Unicode Character Set

- Examples:
  - \u0020 is the *Space* character.
  - \uu0022 is the *Double Quote* character.
  - \uuu0061 is the character 'A'
  - \u000D is the *Carriage Return* character.
  - \u000A is the *Line Feed* character.

- Note:
  The last two characters are called *LineTerminator* (new-line) characters



Language Basic Components
Case Sensitivity and Whitespace

- Java is a *case-sensitive* language

- Examples:
  - Final FINAL final

Keyword

All are different

- Java ignores *spaces*, *tabs*, *new-lines*, and other whitespace, except when it appears within quoted characters and string literals
  - *SpaceCharacter* is equivalent to \u0020.
  - *HorizontalTabCharacter* is equivalent to \u0009 or \t.
  - *FormFeedCharacter* is equivalent to \u000C or \f.
  - *EndOfFileMarker* is defined as \u001A.

- Examples:
  - ' ' " "

Will not be ignored

Language Basic Components
Comments

Java supports three types of comments:

1. A *single-line* comment. It begins with the characters // and continues until the end of the current line.

   Example:

```
int i = 0;          // Initialize the loop variable
```

2. A *multi-line* comment. It begins with the characters /* and continues, over any number of lines, until the characters */

   Example:

```
/*
 * First, establish a connection to the server.
 * If the connection attempt fails, quit right away.
 */
```

Language Basic Components
Comments – doc

3. A special *doc* comment:
   - It begins with /** and ends with */ , cannot be nested
   - It must appear immediately *before* a type or member definition and documents that type or member
   - The documentation can include:
     - Simple HTML formatting *tags* and
     - Other special *keywords* that provide additional information
   - Ignored by the compiler
   - Turned into HTML documentation by the *javadoc* program.

Language Basic Components
Comments – doc

Example:

```
/**
 * Upload a file to a web server.
 *
 * @param   file The file to upload.
 * @return  <tt>true</tt> on success,
 *          <tt>false</tt> on failure.
 * @author  Dr Abdulghani Al-Qasimi
 */
```

Language Basic Components
Comments – doc tags

@author *name*

@version *text*

@param *parameter-name description*

@return *description*

@exception *full-classname description*

@throws *full-classname description*

@see *reference*

@deprecated *explanation*

@since *version*

@serial *description*

@serial include

@serial exclude

@serialField *name type description*

@serialData *description*

Language Basic Components
Keywords (Reserved Words)

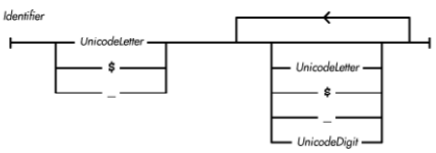
abstract	const	final	int	public	throw
assert	continue	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true
byte	double	goto	new	strictfp	try
case	else	if	null	super	void
catch	enum	implements	package	switch	volatile
char	extends	import	private	synchronized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

Notes:

- Keywords can not be used as identifiers (names for variables, classes … etc).
- const and goto are reserved but aren't actually used in the language
- strictfp was added in Java 1.2,
- assert was added in Java 1.4, and
- enum was added in Java 5.0.

Language Basic Components
Identifiers

- Identifiers are the names a programmer gives to classes, methods, fields, variables and constants
  - An identifier may have any length
  - It may contain Unicode *letters* and *digits*
  - It may not begin with a digit
  - It may not contain punctuation characters, except:
    - The ASCII underscore (_) and dollar sign ($)
    - Other Unicode currency symbols such as £ and ¥.
- Syntax:



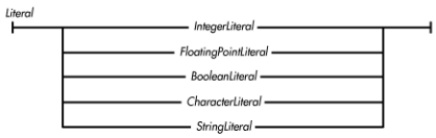
- Examples: i x1 the_current_time isLegal

Java Naming Conventions

- Class names are in UpperCamelCase
- Member names are in lowerCamelCase
- Method names are in lowerCamelCase
- Constant names are in ALL_UPPER_CASE

Language Basic Components
Literals

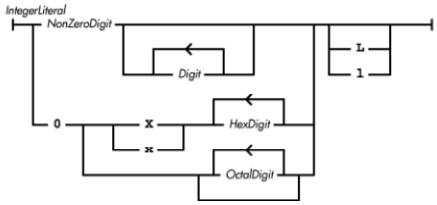
- Literals are values that appear directly in Java source code.
- They include *integer* and *floating-point* numbers, *characters* within single quotes, *strings* of characters within double quotes, and the keywords *true*, *false* and *null*.
- Syntax:



- Examples:
  - 1 1.0 '1' "one"
  - true false null

Language Basic Components
Integer Literals

- Syntax:

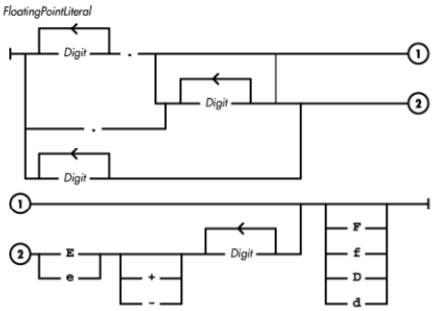


- A *NonZeroDigit* is one of: 1, 2, 3, 4, 5, 6, 7, 8, or 9.
- An *OctalDigit* is one of: 0, 1, 2, 3, 4, 5, 6, or 7.
- Limits:

Representation	Minimum Value	Maximum Value
Hexadecimal	`int 0x80000000` `long 0x8000000000000000L`	`int 0x7fffffff` `long 0x7fffffffffffffffL`
Octal	`int 020000000000` `long 01000000000000000000000L`	`int 017777777777` `long 0777777777777777777777L`
Decimal Equivalent	`int -2147483648` `long -9223372036854775808L`	`int 2147483647` `long 9223372036854775807L`

Language Basic Components
Floating Point Literals

- Syntax:



- Limits:

Representation	Minimum Value	Maximum Value
float	`1.40239846e-45f`	`3.40282347e38f`
double	`4.94065645841246544e-324`	`1.79769313486231570e308`

Language Basic Components
Character & String Literals

- Syntax:

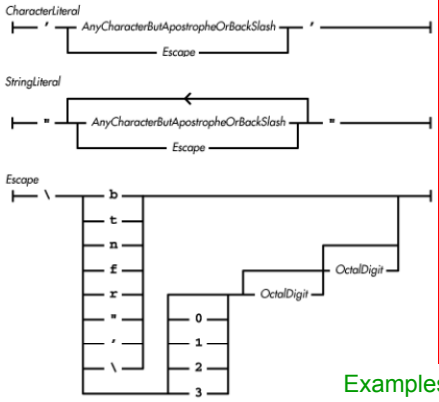


Table of *Escape* Sequences:

Escape Sequence	Unicode Equivalent	Meaning
\b	\u0008	Backspace
\t	\u0009	Horizontal tab
\n	\u000a	Linefeed
\f	\u000c	Form feed
\r	\u000d	Carriage return
\"	\u0022	Double quote
\'	\u0027	Single quote
\\	\u005c	Backslash
\xxx	\u0000 to \u00ff	The character corresponding to the octal value *xxx*

Examples: '\t' "A String" 'A' '\141'

Language Basic Components
Punctuations

Punctuation characters are divided into:

- Separators:
  ( ) { } [ ] < > : ; , . @
- Arithmetic Operators: Numerical operands and results

Operator	P	Action	Examples
++ --	4	Increment / Decrement	x=51; y=23; ➔ x++ ➔ x=52, y-- ➔ y=22
++ --	5	Increment / Decrement	x=51; y=23; ➔ ++y ➔ y=24, --x ➔ x=50
unary + -	5	Unary Plus or Minus	-50, +27, -1
*	6	Multiplication	3 * 16 ➔ 48, 3.0 * 16 ➔ 48.0
/	6	Division	23 / 5 ➔ 4, 23 / 5.0 ➔ 4.6, 25.0 / 5.0 ➔ 5.0
%	6	Modulus	10 % 7 ➔ 3
+ -	7	Addition or Subtraction	10 + 7 ➔ 17, 10 -7 ➔ 3

Language Basic Components
Operators

- Comparison Operators: Comparable operands, boolean result

Operator	P	Action	Example: x=42, y=35	Result
<	9	Less than	x < y	false
<=	9	Less than or equal to	x <= y	false
>=	9	Greater than or equal to	x >= y	true
>	9	Greater than	x > y	true
instanceof	9	Type comparison	Compares a reference variable to a data type	
==	10	Equal to	x == y	false
!=	10	Not equal to	x != y	true

- Logical Operators: Boolean operands, boolean result

Operator	P	Action		Example: a=true, b=false	Result
!	5	Not (unary)		!b	true
&&	14	Conditional and	SC	a && b	false
||	15	Conditional or	SC	a || b	true

Examples:
The instanceof Operator

1. "string" instanceof String // True: all strings are instances of String
2. "" instanceof Object // True: strings are also instances of Object
3. null instanceof String // False: null is not an instance of anything
4. Object o = new int[] {1,2,3};
   a. o instanceof int[ ] // True: the array value is an int array
   b. o instanceof byte[ ] // False: the array value is not a byte array
   c. o instanceof Object // True: all arrays are instances of Object
5. Use instanceof to make sure that it is safe to cast an object

```
if (object instanceof Point)
    Point p = (Point) object;
```

Language Basic Components
Operators

- Bitwise Operators: Integer (all) or boolean (&|^) operands

Operator	P	Action	Example: u=0xF3, v=0x2A	Result
~	5	Bitwise complement (unary)	~u	0x0C
&	11	Bitwise and full	Same as && and ||, but both operands are fully evaluated	
|	13	Bitwise or full		
^	12	Bitwise exclusive-or	u ^ v	0xD9
<<	8	Shift bits left, fill with zeros	v << 2	0xA8
>>	8	Shift bits right, fill with sign bit	u >> 2	0xFC
>>>	8	Shift bits right, fill with zeros	u >>> 2	0x3C

- Conditional Operator:

Operator	P	Action	Example	Result
? :	16	boolean ***exp*** ? v if ***T***: u if ***F***	10 > 30 ? 100 : 200	200

Language Basic Components
Operators

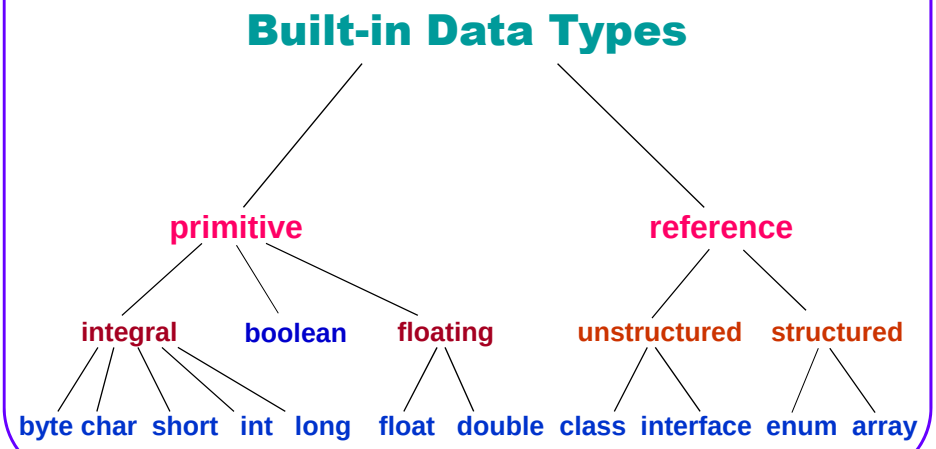
- Assignment Operators: Stores a value to a variable

Operator	P	Action	Example: x=5, y=40, z=101
=	17	Assignment	z = x * y ➔ z =200
+= -= *= /= %= &= |= ^= <<= >>= >>>=	17	Assignment with operation as indicated before for each operator	z %= y ➔ z = 21

- Other Operators:

Operator	P	Action	Example
( )	1	Parentheses	3+(4*5) ➔ 60
+	3	Combines two strings into one	"I " + "love " + "Java" ➔ "I love Java"
(*type*)	5	Type casting / changing type	(int) 4.7 ➔ 4, (double) 37 ➔ 37.0
new	2	Object creation	int[ ] d=new int[ ] {5,9,31} ➔ d➔ {5,9,31}
[ ]	1	Array element access	d[1] ➔ 9
.	3	Object member access	d.length ➔ 3

Language Basic Components
Data Types in Java



Primitive Data Types

- Java supports eight basic (*primitive*) data types
- Primitive data values are directly stored in suitable memory locations
- Details are shown in the following table:

Type	Contents	Representation	Range of values	Default
boolean	----	8-bits, not convertible to any type	`false, true`	`false`
byte	integer	8-bit, signed, two's complement	`-128 to 127`	`0`
short	integer	16-bit, signed, two's complement	`-32768 to 32767`	`0`
int	integer	32-bit, signed, two's complement	`-2147483648 to 2147483647`	`0`
long	integer	64-bit, signed, two's complement	`-9223372036854775808 to 9223372036854775807`	`0`
char	Unicode	16-bit, unsigned, Unicode	`'\u0000' to '\uFFFF'`	`\u0000`
float	real	32-bit, IEEE 754	`1.40239846e-45 to 3.40282347e+38 Same for negative`	`0.0`
double	real	64-bit, IEEE 754	`4.94065645841246544e-324 to 1.79769313486231570e+308 Same for negative`	`0.0`

Primitive Type Conversion

Convert from:	Convert to:							
	boolean	byte	short	char	int	long	float	double
boolean	-	No	No	No	No	No	No	No
byte	No	-	Auto	Cast	Auto	Auto	Auto	Auto
short	No	Cast	-	Cast	Auto	Auto	Auto	Auto
char	No	Cast	Cast	-	Auto	Auto	Auto	Auto
int	No	Cast	Cast	Cast	-	Auto	Auto-	Auto
long	No	Cast	Cast	Cast	Cast	-	Auto-	Auto-
float	No	Cast	Cast	Cast	Cast	Cast	-	Auto
double	No	Cast	Cast	Cast	Cast	Cast	Cast	-

Object Wrappers for Primitives

- Some data structure library classes can only operate on objects, not primitive variables
- The object *wrappers* can easily convert a primitive into the equivalent object, so it can be processed by these data structure classes
- Each of the eight primitive types has a corresponding predefined wrapper class type
- Each of these classes defines static MIN_VALUE and MAX_VALUE variables for its minimum and maximum values
- Example:

```
Java.lang.Integer n;     // wrapper
n=new Integer(275);      // Created
int x;                   // Primitive
n += 50;                 // Autoboxing
x = n.intValue();        // Old, see doc
x = n;                   // Autoboxing
```

Primitive type	Corresponding wrapper class name in java.lang
boolean	Boolean
char	Character
int	Integer
long	Long
byte	Byte
short	Short
double	Double
float	Float

Boxing, Unboxing & Autoboxing

- *Boxing* converts a primitive value to its corresponding wrapper object
- *Unboxing* converts a wrapper object to its corresponding primitive value
- Boxing or unboxing conversion can be explicitly specified a with a *cast*
- Boxing or unboxing conversion can be automatic (called *autoboxing*)
- Autoboxing is done:
  - When a value is assigned to a variable or passed to a method
  - If a wrapper object is used with an operator or a statement that expects a primitive value
- Examples:

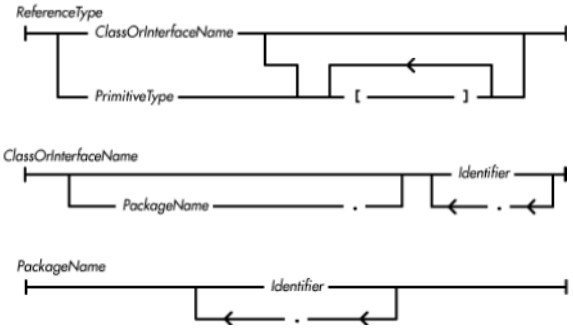
```
Integer i=0;          // int literal 0 is boxed into an Integer object
Number n=0.0f;        // float literal is boxed into Float and widened to Number
Integer i=1;          // This is a boxing conversion
int j=i;              // i is unboxed here
i++;                  // i is unboxed, incremented, and then boxed up again
Integer k=i+2;        // i is unboxed and the sum is boxed up again
i=null; j=i;          // unboxing here throws a NullPointerException
```

Reference Data Types

- Reference data types are *composite* types
- They are composed by the user from other data types:
  - Primitive and/or
  - Composite
- They are created *dynamically* by the new operator
- Java provides five reference data types:
  - arrays,
  - classes,
  - interfaces,
  - enum and
  - Annotation
- The most important reference type is the ***class***.

Reference Data Types

- Syntax:



- Example:
  - java.lang.System // System class provided by Java
  - java.lang.Integer[ ] // Array of the Integer wrapper class

Operations for Reference Data Types

- The only operation for reference−type variables is accessing the referenced object
- No direct access to the underlying memory, including no pointer arithmetic
- The object access operation is performed only implicitly by the following operations:
  - An expression accessing a variable or method of a class or interface object
  - An expression accessing an element of an array object
  - A type comparison using the instanceof operator

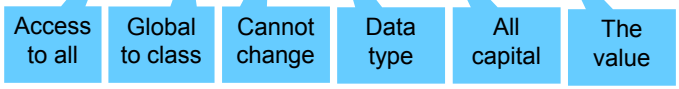
Variables & Constants

- Symbolic names (*identifiers*) are used to represent storage locations for the data values being used in a running program
  - Variable: if its value can change during execution
  - Constant: if its value can not change during execution
- Variables and constants must be *declared* before they can be used
- *Declaration* tells the Java compiler:
  1. What name the variable or constant should be called by
  2. What data type it will hold.

Constant Declaration

- Constants in Java are declared inside a class and outside all methods
- Form:

```
public  static  final  type  NAME  = literal;
```



- Example:

```
public static final double PI = 3.14159;
```

Variable Declaration

- Variables in Java are declared inside a class, inside methods and in inside any block of code within a method
- Syntax:
  [scope_modifier] [usage_modifiers] *type variable_name* [= initializer];
- Examples:

```
private double x, y;
public int [ ] buffer;
public static double maxWage = 30000;
```

Variable Modifiers

- Scope modifiers:

Modifier	Within Same Class	Within Same Package	In a Subclass	Everything else
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

- Usage modifiers:

Modifier	Variable Kind	Example
static	Global class field	public static int maxAge = 65;
final	A constant	public static final double PI = 3.14;

Field Variables

- A Java program may have three kinds of variables:

1. Class field-variables:
   - Declared inside the class, outside of any method
   - Declared as *static* fields of the class
   - Associated with the class itself
   - Global to all *instances* and *methods* of the class
   - Exist even when there are no instances of their class
   - Automatically initialized to a default value

Instance Variables

2. Class Instance-variables:
   - Declared inside the class, outside of any method
   - Declared as *non-static* fields of the class
   - Associated with an *instance* of the class
   - Global to all *methods* of the class *instance*
   - Exist as long as their associated *instance* exists
   - Automatically initialized to a default value

Field and Instance Variables

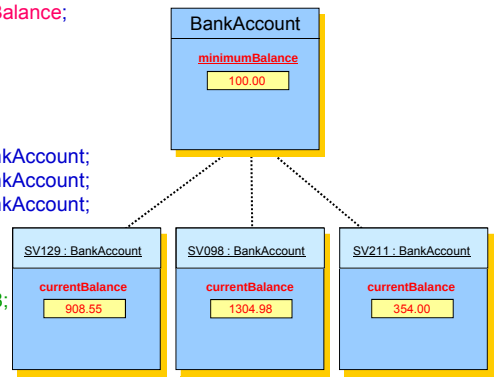
- Example:

```
public class BankAccount {
   private static double minimumBalance;
   private double currentBalance;
       ..............

   minimumBalance = 100;

   BankAccount sv098 = new BankAccount;
   BankAccount sv129 = new BankAccount;
   BankAccount sv211 = new BankAccount;
       ..............

   sv129.currentBalance=908.55;
   sv098.currentBalance=1304.98;
   sv211.currentBalance=354.00;
}
```



Local Variables

3. Method Local-variables:
   - Declared anywhere within a *block* of code inside of a function (*method*)
   - Associated with the *block* they were declared in
   - Formal parameters of a function are local to it
   - Exist only when their block is being executed
   - Not automatically initialized
   - Syntax:

     [final] type variable_name [= initializer];

   - Examples:

     final double PI = 3.14;

     for(int i = 0; i < 10; i+ +)

Primitive vs. Reference variables

- Think of a reference as an *address* of the location where an object is stored in memory
- When a variable is declared, then:
  - If the variable holds primitive-type data,
    1. The compiler will reserve (*allocate*) enough space in memory to hold the value of the variable
    2. The name of the variable will refer to its stored value

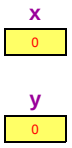
Primitive vs. Reference variables

- If the variable holds reference-type data, then:
  1. The compiler will *allocate* enough space in memory to hold only the address of another location in memory, where the value can be stored later
  2. The compiler will not reserve (*allocate*) memory space for the value of the variable
  3. The name of the variable will refer to its stored location (*reference*) and not to its value
  4. To allocate space in memory for a reference object it must be dynamically created by the new operator.

Primitive vs. Reference variables

Example:

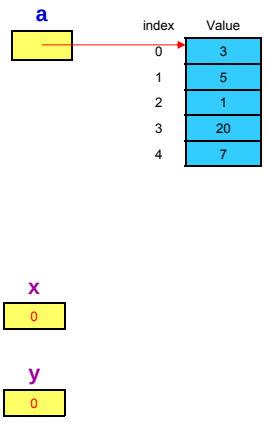
```
int x, y;
```



Primitive vs. Reference variables

Example:

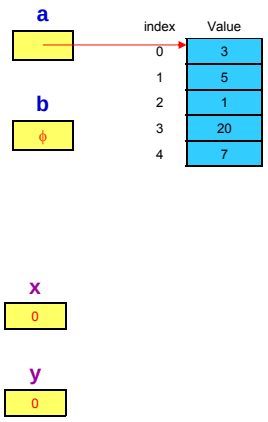
```
int x, y;
int [ ] a = {3,5,1,20,7};
```



Primitive vs. Reference variables

Example:

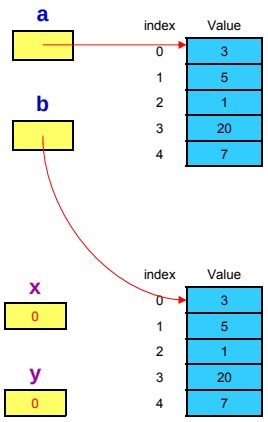
```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
```



Primitive vs. Reference variables

Example:

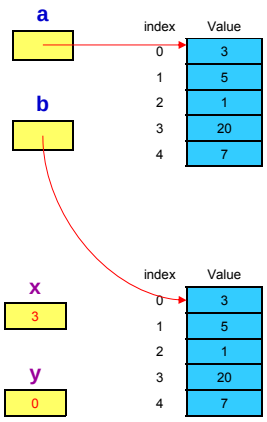
```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
```



Primitive vs. Reference variables

Example:

```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
x = 3;
```

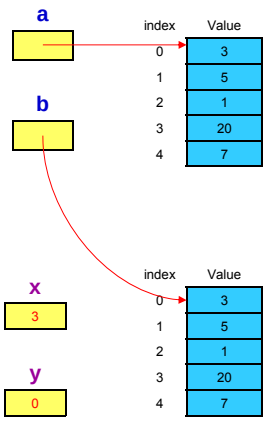


Primitive vs. Reference variables

Example:

```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
x = 3;
```

x == y? ➔ false

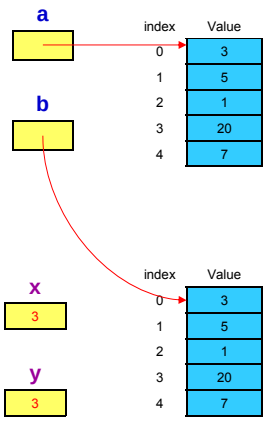


Primitive vs. Reference variables

Example:

```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
x = 3;

x == y? ➔ false
y = x;
```



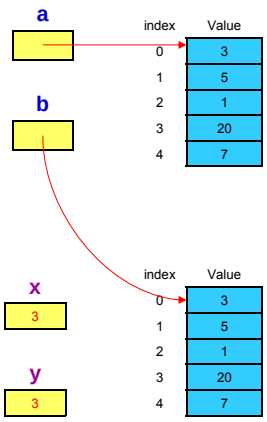
Primitive vs. Reference variables

Example:

```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
x = 3;

x == y? ➔ false
y = x;
x == y? ➔ true
```

values are copied and compared



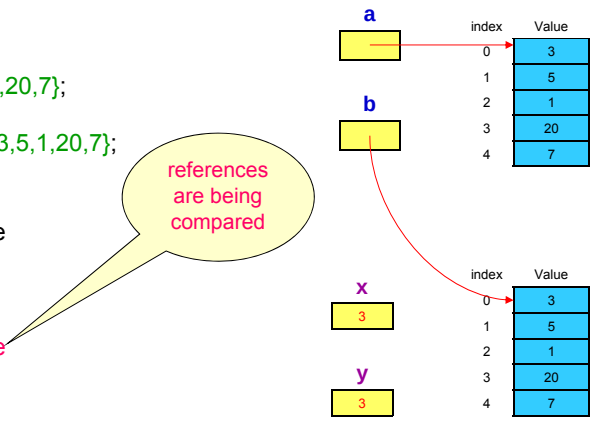
Primitive vs. Reference variables

Example:

```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
x = 3;

x == y? ➔ false
y = x;
x == y? ➔ true

a == b? ➔ false
```



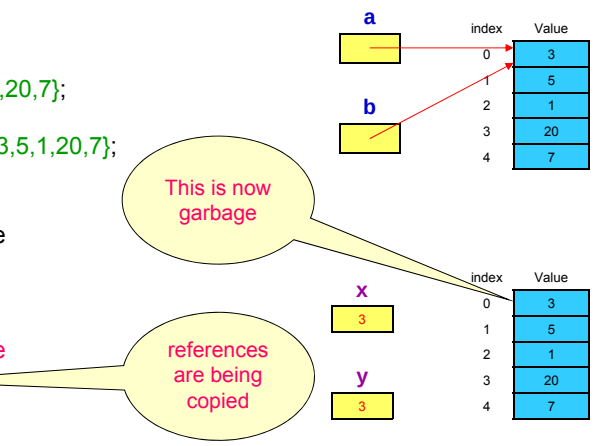
Primitive vs. Reference variables

Example:

```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
x = 3;

x == y? ➔ false
y = x;
x == y? ➔ true

a == b? ➔ false
b = a;
```



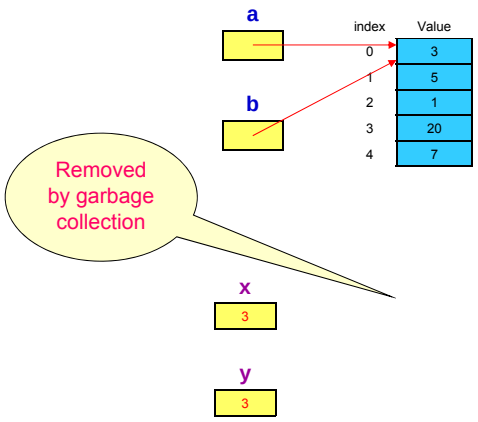
Primitive vs. Reference variables

Example:

```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
x = 3;

x == y? ➔ false
y = x;
x == y? ➔ true

a == b? ➔ false
b = a;
```



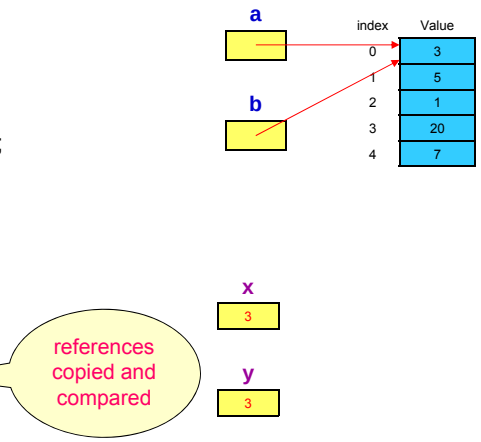
Primitive vs. Reference variables

Example:

```
int x, y;
int [ ] a = {3,5,1,20,7};
int [ ] b;
b = new int [ ] {3,5,1,20,7};
x = 3;

x == y? ➔ false
y = x;
x == y? ➔ true

a == b? ➔ false
b = a;
a == b? ➔ true!
```



Issues of Using References: 1. Aliases

- Aliases:
  More than one name for the same object
- Example:
  a and b are two names for the same array. So:

```
System.out.println(a[3]); ➔ prints 20
b[3] = 9;
System.out.println(a[3]); ➔ prints 9
```

Issues of Using References: 2. Type Casting

- The value null is the default value for any reference variable
- It can be assigned to any reference variable without a type cast
- Java **does not allow** reference types to be **cast** to primitive data types nor the reverse

Issues of Using References: 2. Type Casting

- Java reference types form a *class hierarchy*.
- Every Java reference type extends some other type, known as its *superclass*.
- The *superclass* of all java classes is called Object; All Java classes extend Object directly or indirectly
- An object cannot be converted to an unrelated type.

Issues of Using References: 2. Type Casting

1. Widening Conversion:
   An object can be converted to the type of its *superclass* or of *any ancestor* class; No explicit cast is required.

2. Narrowing Conversion:
   An object can be converted to the type of a *subclass*, but an explicit cast is required. (Check validity before casting or error may occur)

   Example:

```
String s, t;
Object o;
s = "conversion test";
o = s;              // Assign a String to an Object-type variable
t = (String) o;     // Cast back to a String type
```

Issues of Using References: 2. Type Casting

- Any *array* can be converted to an *Object* value through a widening conversion.
- A narrowing conversion with a cast can convert such an *Object* value back to an *array*.
- An *array* can be converted to another type of *array* if:
  - The *base types* of the two arrays are reference types and
  - The *base types* themselves can be converted.
- An array of *primitive* type cannot be converted to any other array type, even if the primitive base types can be converted.

Issues of Using References: 3. Copying Objects

To make a full copy of an object use the special **`clone()`** method, automatically inherited by all objects from java.lang.Object.

Examples:

```
Point p = new Point(1,2);          // p refers to one object
Point q = (Point) p.clone( );      // q refers to a copy of p
q.y = 42;                          // Modify the new object

int[] data = {1,2,3,4,5};          // An array
int[] copy = (int[]) data.clone( ); // A copy of the array
```

Issues of Using References: 3. Copying Objects

Notes:

1. A cast is necessary to make the return value of the `clone()` method the correct type.
2. Java only allows an object to be cloned if the object's class has explicitly declared itself to be *cloneable* by implementing the Cloneable interface.
3. Arrays are always cloneable.
4. The inherited clone() method makes a shallow copy of the object.
5. To make a deep copy of an object, the class may need to override the `clone()` method.

Issues of Using References: 4. Shallow vs. Deep Copy

Shallow Copy:

- An operation that copies a class instance to another by:
  1. Making a duplicate copy of all primitive values
  2. Making a duplicate copy of all references, but not their values.
- Results are the two instances contain duplicate references to same values.

Deep Copy:

- An operation that copies a class instance to another by:
  1. copying from the source element by element
  2. Making a duplicate copy of all primitive values
  3. Using observer methods as necessary to eliminate nested references
  4. Copying the primitive types that references refer to.
- Results are the two instances do not contain any duplicate references.

Issues of Using References:
5. Comparing Objects for equality

- Comparing the equality of objects isn't easy!
- When working with reference types, there are two kinds of equality:
  1. Equality of reference and
  2. Equality of object
- All objects inherit an `equals()` method from java.lang.Object.
- The default implementation of the `equals()` method simply uses `==`
  - For primitive types, it tests the equality of their values
  - For reference types, it tests the equality of references.
- To compare for equality of objects, the objects class can define its own version of the `equals()` method.
- Arrays can be compared for equality by using the method java.util.Arrays.equals( ).

Issues of Using References:
6. Comparing Objects in general

- Comparing two objects in general, requires their class to implement java.lang.Comparable, interface or to provide a Comparator object to compare them.
- All objects inherit an `compareTo()` method from java.lang.Comparable.
- The default implementation of the `equals()` method simply uses `==`
  - For primitive types, it tests the equality of their values
  - For reference types, it tests the equality of references.
- To compare for equality of objects, their class can define its own version of the `equals()` method.
- Arrays can be compared for equality by using the method java.util.Arrays.equals( ).

Issues of Using References: 6. Printing Objects

- To facilitate printing of an object, it must be converted to a String
- Java provides two ways to do that:
  1. Use the appropriate `toString()` method:
     - It is automatically inherited by all objects from java.lang.Object.
     - It returns a string associated with the object.
     - Any class can override `toString()` to provide a deep conversion
  2. Perform an implicit cast via the string concatenation operation (+):
     - Any time a string is concatenated with any object or base type, that object or base type is automatically converted to a string.
- Examples:

Wrong Conversion	Correct Conversion
String s = (String) 4.5;	String s = " " + 4.5; \\ poor style
String t = "Value = " + (String) 13;	String t = "Value = " + 13;
String u = 22;	String u = Integer.toString(22);

Issues of Using References: 7. Parameter Passing

All Java arguments are *passed by value*

- If the variable is of a primitive type, its actual value is passed to the method.
- If it is a reference type, then the reference that it contains is passed to the method.
  (It is like passing its contents by reference)

Issues of Using References: 8. Garbage Collection

- The *new* keyword creates a new object (instance) of a reference type
- Java automatically allocates whatever amount of memory is necessary to store it in the heap
- Any object is considered *garbage* when there are no references to it stored in any variables, fields of any objects or elements of any arrays
- Java has a *garbage collection* process that automatically reclaims any garbage memory for reuse.
- Example:

```
Point p = new Point(1,2);              // Create a Point object
double d = p.distanceFromOrigin( );    // Use it for something
p = new Point(2,3);                    // Create a new object. First reference is lost
```

Expressions

- Primary expressions consist of *literals* and *variables*
- Examples:
  - 1.7 // A floating-point literal
  - true // A boolean literal
  - sum // A variable
- More complex expressions are made by using *operators* to combine primary expressions
- Examples:
  - sum = 1.7
  - sum = 1 + 2 + 3*1.2 + (4 + 8)/3.0
  - sum/Math.sqrt(3.0 * 1.234)
  - (int)(sum + 33)

Expression Statements

- A statement is a *single command* executed by the Java interpreter
- Statements are *run in sequence* of same order as they were written in the program
- Expression statements are called *side-effect* statements because they affect the program state in some way.
- Legal types of expression statements are: assignments, increments and decrements, method calls, and object creation.
- Many statements are *flow-control* statements that change the order of execution in well-defined ways:

Flow-Control Statements

- These statements can be sub-divided into:
  - Conditional Statements:
    if and switch
    They work in Java similar to the way they work in other languages
  - Loop Statements:
    while, for, do-while and foreach – see Sec.6.3.2
    They work in Java similar to the way they work in c++ except that they strictly accept only *boolean* expressions and conditions
  - Explicit flow-control statements:
    return, break and continue
    They work in Java similar to the way they work in c++

Other Statements

- The Compound Statement:
  Defines a statement block where one or more statements are grouped together between the braces "{" and "}"

  A block can include:
  - Local variable declarations
  - Other blocks (nested)

- Error checking and handling statements:
  Used to throw error exceptions in some methods (throw), catch those errors in some error handling methods (try) and to check for the validity of a certain condition (assert)

Statements: Syntax Summary

Legend: *italics* → must be supplied by user, normal → keyword, [options] → optional items, … → repeat, *var* → variable name, *expr* → expression, *label* → identifier, *error* → error-code

Statement	Purpose	Syntax

expression	Side effects	*var* = *expr*; // assignments
		expr++; // increments / decrements
		method( ); // method calls
		new *Type*( ); // object creation

compound	Statement block	{ *statements* } // substitutes for any statement
empty	Do nothing	;
labeled	Name a statement	*label* : *statement* // row: for (r=0; r<10; r++) {
variable	Declare local variable	[final] *type name* [= *value*] [, *name* [= *value*]] ...;
if	Conditional	if (*boolean-expr*) *statement* [else *statement*]

switch	Conditional	switch (*expr*) { // using break prevents fall thru
		[case *expr* : *statements*] ...
		[default : *statements*]
		} // break ends the switch

Statements: Syntax Summary

Statement	Purpose	Syntax
while	Loop	while (*boolean-expr*) *statement*
do	Loop	do *statement* while (*boolean-expr*);
for	Simplified loop	for (*init* ; *condition* ; *increment*) *statement*
for/in	Collection iteration or called "Foreach"	for (*variable* : *iterable*) *statement*
break	Exit block	break [*label*] ;
continue	Restart Loop	continue [*label*] ;
return	End method	return [*expr*] ;
synchronized	Critical section	synchronized (*expr*) { *statements* }
throw	Throw exception	throw *expr* ;
try	Handle exception	try { *statements* } [catch (*type name*) { *statements* }] ... [finally { *statements* }]
assert	Verify invariant	assert *invariant* [: *error*] ;

Methods

- A Java *method* defines a group of statements that perform a particular operation
- `static` indicates a ***static*** or ***class*** method
- A method that is not `static` is an ***instance*** method
- All method arguments are ***call-by-value***
  - Primitive type:
    value is passed to the method. Method may modify the local copy ***but*** will not affect caller's value
  - Object reference:
    address of object is passed to the method. Change to the reference variable does not affect caller, ***but*** operations can affect the caller's object.

Functions

- Java does not allow non-member functions
- All functions in Java are actually *methods*
- Methods that are declared static, are equivalent to functions (i.e. they belong to the class, but not to a particular object)
- Example:

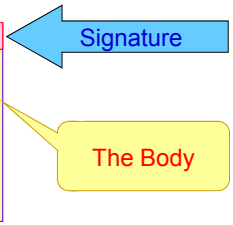
```
static int factorial(int n) {
    int result = 1;
    for (int i = n; i > 0; i--) {
        result *= i;
    }
    return result;
}
```

Defining Methods

- A method consists of two parts:
  1. The method signature
  2. The method body: An arbitrary sequence of statements enclosed within curly braces.
- Syntax:

  modifiers type name (*paramlist*) [throws *exceptions*]
  { *The Method Body Statements;* }
- Example:

```
public static int max(int [ ] data)
{
    int max = data[0];
    for (int i=1; i < data.length; i++) {
        if (data[i] > max) max = data[i];
    }
    return max;
}
```



The Method Signature

- A method *signature* is the specification that defines everything we need to know about a method before calling it
- That specification includes:
  - The name of the method
  - The number, order, type, and name of the *parameters* used by the method
  - The type of the value *returned* by the method
  - The checked *exceptions* that the method can throw
  - Various method *modifiers* that provide additional information about the method
  - (Exceptions and Modifiers are not part of the signature)

Method Modifiers

Modifier	Meaning
abstract	The method is a specification without implementation. No body is provided for the method; it is provided by a subclass. The signature is followed by a semicolon. The enclosing class must also be abstract
final	The method may not be overridden or hidden by a subclass. All private methods and methods of a final class are implicitly final
native	The method implementation is written in some "native" language such as C and is provided externally to the Java program. No body is provided; the signature is followed by a semicolon
Private, protected, public	Work the same as the scope modifiers defined earlier for field variables

Method Modifiers (cont.)

Modifier	Meaning
static	This is a class method associated with the class itself rather than with an instance of the class. It cannot use any instance methods or fields. It is not passed an implicit *this* object reference. It can be invoked through the class name
strictfp	Must perform floating-point arithmetic using 32- or 64-bit floating point formats, strictly according to IEEE 754 standard
synchronized	Makes a method thread-safe; It prevents two threads from executing the method at the same time

Exceptions

- Exceptions are Throwable objects of *unexpected* events during program execution
- There are two main types of exceptions:
  - Specified by the Error subclass
  - Specified by the Exception subclass
- Java exception-handling distinguishes between *checked* and *unchecked* exceptions:
  - Any exception object that is an Error is unchecked
  - Any exception object that is an Exception is checked
  - Any exception object that is a subclass of java.lang.RuntimeException is unchecked

Exceptions

- An unchecked exception is an *unexpected error* related to the program run-time environment like running out of memory
- A checked exception is an *expected exception*, so the compiler checks to make sure that it is declared in method signatures
- The compiler produces a compilation error if it was not declared
- If a method calls another method that can throw a checked exception, the calling method must either:
  - Include exception-handling code to handle that exception, or
  - Use throws to declare that it can also throw that exception.

Exception Declaration Example

- Example:

```
// This method reads the first line of text from a named file
public static String readFirstLine(String filename) throws IOException
{
    BufferedReader in = new BufferedReader(new
    FileReader(filename));

    String firstline = in.readLine( );          // Can cause IOException
    in.close( );
    return firstline;
}
```

Exception Handling

- There are two parts to exception handling:
  1. Throwing the exception: Must be in a method that is declared to throw this exception

     Example:

     ```
     Exception ex = new Exception ("Something is really wrong.");
     throw ex;
     ```

  2. Catching the exception: Must be in the method that caused the exception, or in a calling method that was passed the exception from the called method

     Example:

     ```
     try {
        y = 0;
        x = 10 / y;
     } catch (Exception ex) {
        System.out.println(ex.getMessage());
     }
     finally {
        // Code that always gets executed
     }
     ```

Empty Slide

JAVA Language Review

Arrays

Array's Abstract Definition

An array is a finite ordered set of homogeneous elements.

- All the elements of an array have the same size
- The ordering of elements is defined by a positional index
- All array operations involve accessing an array element
- Array elements are accessed by their position

Arrays in Java

- Java arrays are structured composite objects
- Java hides the array implementation details
- Java arrays can grow or shrink upon reassignment
- Java does not support matrix-style multi-dimensional arrays
- Java supports arrays of arrays providing some multi-dimensional capability

Arrays in Java: Declaration forms

- Single-Dimensional array:

```
// Declaration, Instantiation, and initialization in one step
base-type array-name [ ] = {value-list};   // C-style w/ initializer
base-type [ ] array-name = {value-list};   // Java style w/ initializer

base-type [ ] array-name;                  // Declaration only
array-name = new base-type [size];         // Instantiation

// Declaration and instantiation with default initialization
base-type [ ] array-name = new base-type [size];
```

Arrays in Java: Declaration forms

- Multi-Dimensional array:

  base-type [ ] [ ] … [ ] array-name *= {{value-list},{value-list}, … {value-list}};*

  base-type [ ] [ ] … [ ] array-name;
  array-name = new base-type [$size_1$][$size_2$] … [$size_n$];

  base-type [ ] [ ] … [ ] array-name = new base-type [$size_1$] [$size_2$] … [$size_n$];

- Note:
  Only the leftmost dimension or dimensions must be specified.

- Example:
  ```
  float[ ][ ][ ] globalTemp = new float[360][ ][ ];
  float[ ][ ][ ] globalTemp = new float[360][180][ ];
  ```

Arrays in Java

- Arrays are allocated (instantiated) using the *new* keyword
- Default initialization:
  - Primitives are initialized to zero
  - The boolean type is initialized to false
  - References are initialized to null
- Java array types are not classes, but array instances are objects
- Arrays inherit the methods of java.lang.Object
- Arrays have a public final int field named length that specifies the number of elements in the array, indexed from 0 to array-name.length-1
- Arrays implement the *Cloneable* interface and override the clone( ) method to guarantee cloning without errors
- The array index expression must be of type int, or a type that can be widened to an int: byte, short, or even char.

Array Examples

- Examples:

```
String[ ] greetings = { "Hello", "Hi", "Howdy" };

int[ ] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};

// Initializer expressions are computed at runtime
Point[ ] points  =  {  circle1.getCenterPoint( ),
                       circle2.getCenterPoint( ) };

int[ ][ ] products = {  {0, 0, 0, 0},
                        {0, 1, 2, 3},
                        {0, 2, 4, 6},
                        {0, 3, 6, 9} };
```

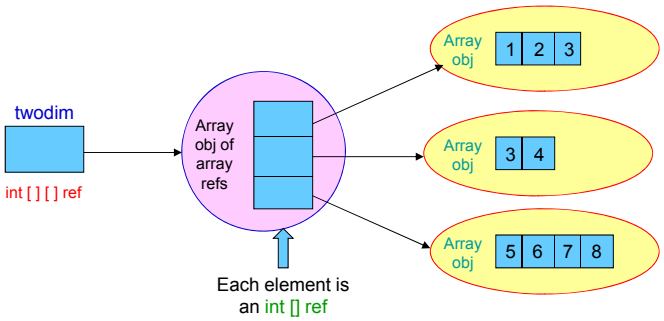
```
int[ ][ ] products = {  {0},
                        {0, 1},
                        {0, 2, 4},
                        {0, 3, 6, 9} };
```

A jagged Array

Array Examples

- The following declares and initializes a two-dimensional array which is not rectangular:

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



Array Examples

- Examples:

```
// Array reference & array index may be complex expressions
double datum = data.getMatrix( ) [data.row( )*data.numCol( ) +
                                  data.col ( )];

// Iterating an array using for loop
int [] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
int sumOfPrimes = 0;
for (int i = 0; i < primes.length; i++) sumOfPrimes += primes[i];

// Iterating the same array using foreach loop
for (int p : primes) sumOfPrimes += p;
```

Copying Arrays

Shallow Copy:

- Use the java.lang.Object.clone() method

Example:

```
int[ ] data = { 1, 2, 3 };
int[ ] copy = (int[ ]) data.clone( );
```

Copying Arrays

Shallow Copy:

- Use the java.lang.System.arraycopy( ) method
  - The method takes five arguments:
    - Source array name
    - Source copying start index
    - Destination array name
    - Destination placement start index
    - The number of elements to be copied

Example:

```
// Copy n-1 elements of array a to itself
// Shifting up one position
System.arraycopy(a, 1, a, 0, n-1);
```

Array Utilities

- The java.util.Arrays class contains a number of static utility methods for working with arrays
- Works for primitive type arrays and arrays of objects
- Provided methods include:
  - The sort( ) and binarySearch( ) for sorting and searching
  - The equals( ) for comparing the content of two arrays
  - The Arrays.toString( ) for converting array content to a string
  - The deepEquals( ) and deepToString( ) methods work correctly for multidimensional arrays.
- Example: See ArrayTest.java

JAVA Language Review

Classes

Classes and Objects

- The *class* is the unit of programming in Java
- A Java program is a *collection of classes*
  - Each class definition is in its own `.java` file
  - *The file name must match the class name*
- A class describes *objects* (*instances*)
  - It describes their common characteristics
  - All the instances have these same characteristics
- These characteristics are the *members* of the class

The Class Data Type

- The class defines a unit of *encapsulation* of its members
- Members of a class are:
  - Data elements (i.e. variables) of different types, called *fields*
  - Functions that initialize and manipulate the data elements, called *methods*.
- Syntax:

```
[class_modifiers] class ClassName
   [extends Class_Or_Interface_Name]
   [implements Class_Or_Interface_NameList] {

   Class_Member_Declarations;
}
```

Class Modifiers

- Class modifiers:

Modifier	Meaning
abstract	The class contains unimplemented methods and cannot be instantiated
final	The class cannot be subclassed
public	The class can be accessed, instantiated or extended anywhere its package is located
Non public	A non-public class is accessible only in its package
strictfp	All methods of the class are implicitly strictfp
static	An inner class declared static is a top-level class, not associated with a member of the containing class

Object Literals

- Java defines a literal syntax for special reference type objects:
  - String literals:
    - The data type used to represent text is the **`String`**
    - Strings in Java are objects
    - A String literal is a text in double quotes
    - String literals cannot contain comments
    - String literals may consist of only a single line
    - Example:

```
String name = "Sami";
```

Object Literals

- Class Type literals:
  - Instances of the class named **`Class`** represent Java data types
  - To include a **`Class`** object literally in a Java program, follow the name of any data type with .class
  - Example:

```
Class typeInt = int.class;
Class typeIntArray = int[ ].class;
Class typePoint = Point.class;
```

- The null reference:
  - A literal value that represents reference to nothing
  - The null value is is a member of every reference type
  - Example:

```
String s = null;
```

Constructors

- A constructor is a method whose name is the same as the name of its class
- It serves to perform any necessary initialization for a new object
- Every class in Java has at least one constructor, called the *default constructor*, which has no arguments
- A class can have many constructors
- The default constructor can be overloaded
- Each constructor must have a different signature (parameter list)
- A constructor is declared without a return type, not even void
- A constructor may include a return statement without any value

The `this` Reference

- When a new instance of a class is created:
  - The new object is passed *implicitly* a **`this`** reference to itself
  - The new object is passed *explicitly* all the arguments specified between the constructor's parentheses
  - The constructor uses the **`this`** reference to distinguish between a method parameter and an instance field of the same name
  - The body of the constructor should initialize the **`this`** object
- The **`this()`** call can be used from a constructor to invoke one of the other constructors of the same class, but it can appear only as the first statement in a constructor

Example:
The Point Class

```
/**
 * Represents a Cartesian (x,y) point
 */

import java.lang.Math;

public class Point {
    public double x, y;            // The coordinates of the point

    // A constructor to initializes the point
    public Point (double x, double y) {
            this.x = x;
            this.y = y;
    }

    // Computes the distance of this point from the origin
    public double distanceFromOrigin( ) {
            return sqrt(x*x + y*y);
    }
}
```

Example:
The Date Class

```
public class Date {

    private int month;
    private int day;
    private int year;

    public Date(int month,
                int day,
                int year) {
        this.month = month;
        this.day = day;
        this.year = year;
     }

    public int getMonth() {
        return this.month;
    }

    public int getDay() {
       return this.day;
    }

    public int getYear() {
        return this.year;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public void setDay(int day) {
        this.day = day;
    }

    public void setYear(int year) {
        this.year = year;
    }

}
```

Initializers – Instance Fields

- Field variables are automatically initialized to default values of their types
- If other than default values are needed, then an initializer is required
- Instance Fields:
  The Java compiler generates instance-field initialization code automatically and puts it in the constructor(s) of the class in the same order of the fields

Example:

```
public class TestClass {
   public int len = 10;
   public int[ ] table = new int[len];
   public TestClass() {
      for(int i = 0; i < len; i++) table[i] = i;
   }
}
```

The generated constructor:

```
public TestClass() {
   len = 10;
   table = new int[len];
      for(int i = 0; i < len; i++) table[i] = i;
}
```

If a constructor begins with a `this()` call to another constructor, the field initialization code is inserted in the called constructor

Initializers – Class Fields

- Class fields:
  - They belongs to the class. Must be initialized before calling the constructor
  - The compiler generates a *class initialization method* automatically for every class
  - This is a hidden method that is invoked exactly once before the class is first used
  - The compiler-generated class-field initialization code is inserted into this method
- Arbitrary initalization code for instance fields is written in the constructor
- Arbitrary initialization code for class fields can not be written in the class initialization method because it is hidden
- Java allows doing this through a construct known as a *static initializer*.

Initializer blocks

Static Initializer Block:

- Consists of the keyword static followed by a block of code
- Can appear in a class definition where a field or method definition can appear
- Example:

```
public class TrigCircle {              // Static lookup tables and their initializers
  private static final int NUMPTS = 500;
  private static double sines[ ] = new double[NUMPTS];
  private static double cosines[ ] = new double[NUMPTS];
  static {                             // Statc initializer that fills in the arrays
    double x = 0.0;
    double delta_x = (Circle.PI/2)/(NUMPTS-1);
    for (int i = 0, x = 0.0; i < NUMPTS; i++, x += delta_x) {
      sines[i] = Math.sin(x);
      cosines[i] = Math.cos(x);
    }
  }
}
```

Initializer blocks

- A class can have any number of static initializers
- The body of each initializer block is inserted into the class initialization method, along with any static field initialization expressions
- A static initializer is like a class method in that it cannot use the `this` keyword or any instance fields or instance methods of the class

Initializer blocks

Instance Initializer Block:

- Like a static initializer block, but it initializes an object, not a class
- Like a static initializer block, but it doesn't use the static keyword
- Used to initialize arrays or fields that require complex initialization
- A class can have any number of instance initializers
- Can appear where a field or method definition can appear
- The body of each instance initializer is inserted at the beginning of every constructor for the class, with any field initialization expressions
- Example:

```
private static final int NUMPTS = 100;
private int[ ] data = new int[NUMPTS];
{ for (int i = 0; i < NUMPTS; i++) data[i] = i; }
```

Sources for Classes

- The Java Class Library:
  A class library that includes hundreds of useful classes.
- User-built:
  Users can create the needed classes, possibly using pre-existing classes in the process.
- Off the shelf:
  Software components, such as classes or packages of classes, which are obtained from third party sources. The net.datastructures package is an example

The Java Class Library

- Java has a small core and an extensive collection of packages known as Java API or the Java class library
- The Java class library provides a hierarchy of classes, many of which can be extended by the users
- These classes include some fundamental predefined class types
- The two most important class types are:
  - Object: The *superclass* of all other classes
  - String: Allows the creation and manipulation of string data

The Java Class Library

- A ***package*** consists of some related Java classes
- Java has predefined many useful packages
- Examples:
  - **swing**: A GUI (graphical user interface) package
  - **awt**: Application Window Toolkit (more GUI)
  - **util**: Utility data structures (important to EE367!)

Most Useful Classes: The `Object` Class

java.lang.Object:

- The class *java.lang.Object* is the *ultimate parent* of every other class in the system
- Has methods which can be called on by any and all objects
- Most important methods:

```
public class Object {
    public java.lang.Object();      // Constructor
    public java.lang.String toString();
    public boolean equals(java.lang.Object);
    public int hashCode();          // Makes object usable in a hash table
    public Object clone();          // Copy an object – must be cloneable
}
```

Most Useful Classes: The `String` Class

java.lang.String:

- It is a class type, used when String instances are needed to store some characters in a sequence or for human-readable I/O
- String objects hold a series of adjacent characters, similar to an array
- Strings have methods to:
  - Extract substrings,
  - Convert into lower case,
  - Search a String,
  - Compare two Strings,
  - and so on.
- Arrays of char have none of the methods of the String.
- literals:
  A string literal is zero or more characters enclosed in double quotes

String Examples

- ```
  // String concatenation
  String school = “Taif ";
  school = school + "University";    // school is “Taif University”
  ```

- ```
  // String comparison
  String car = "Bisons";
  if (car == "Nissan")                    // wrong way to compare strings
  if (car.equals("Nissan"))                    // true
  if (car.equalsIgnoreCase("NISSAN"))          // true
  if (car.compareTo("Nissan") == 0)            // true
  ```

- ```
  System.out.println(car.substring(2, 5));        // Prints “ssa"
  ```

Java `String` Properties

- Strings in Java are immutable. (once created, it can not be modified)
- A new String is created on all operations
- Be sure to assign result back into variable, otherwise result of operation is lost
- Length obtained by calling .length() method
- Use the .charAt(i) method to get the character at position i.
  ```

String Examples

```
// My birthday Oct 12, 1973
java.util.Calendar c;
c = new java.util.GregorianCalendar(1973, 10, 12);
String s = String.format("My birthday: %1$tb %1$te, %1$tY", c);
```

```
// Mutable string
StringBuffer buffer = new StringBuffer("two ");
buffer.append("three ");
buffer.insert(0, "one ");
buffer.replace(4, 7, "TWO");
System.out.println(buffer);     // Prints "one TWO three"
```

Packages in Java

- Since each Java class must be in a separate file matching its name, working on a large project can involve a big number of files that must be properly organized
- Java provides a way to organize code into packages
- Java classes need not be explicitly packaged, in which case the default package will be used (but not recommended)
- A package name consists of lowercase letters & numbers
- Multiple words are usually separated by periods (.)

Packaging a Class

- To package a class…
  - The first line in the Java file should be a package declaration in the following format:

```
package foo.bar.baz;
```

  - The class file should be placed in a directory structure matching the package name:

```
foo/bar/baz/Foo.java
```

Using a Package

- Using a file in a package:
  - Use the import keyword as follows, drop the file extension:

```
import java.util.Date;
```

- The import statement tells the compiler to make available classes and methods of another package

JAVA Language Review

Simple Input & Output

Java I/O Streams

- Java defines abstractions for all the possible I/O types
- The main I/O abstractions are called *streams*
- Other I/O abstractions include, *files* and *channels*
- A stream represents a source of input or a destination for output
- There are two broad categories of data:
  - Machine-formatted data – Communicates data between computers
  - Human-readable data – Communicates data between humans and computers
- Accordingly, Java has two broad categories of streams:
  - Byte streams – for machine-formatted data
  - Character streams – for human-readable data

Java Standard Input & Output

- Java has many predefined classes that represent streams of each type:
  - For reading and writing character data, the main classes are the abstract classes Reader and Writer
  - For reading and writing byte data, the main classes are the abstract classes InputStream and OutputStream
- The Java predefined standard input stream, System.in and output stream, System.out, are byte streams, not character streams
- System.in is a static stream that belongs to the InputStream class
- System.out is a static stream that belongs to the PrintStream class
- PrintStream is a subclass of the FilterOutputStream, which is a subclass of OutputStream

Java Standard Output

- System.out provides the following common standard output methods to output multiple lines of text to the standard output window:
  - print(), println(), printf(), …
- System.out.print() prints output parameters in one line
- System.out.println() prints output parameters with end line
- System.out.printf() prints output parameters in a C-printf()-like formatted text

The DecimalFormat Class

- Use a DecimalFormat object from java.text.* to format numerical output
- Example:

```
double num = 123.45789345;
DecimalFormat df = new DecimalFormat("0.000");
//three decimal places

System.out.print(num);          ---> 123.45789345
System.out.print(df.format(num)); ---> 123.458
```

Java Standard Input

- System.in can only input a single byte directly
- To input primitive data values, use the java.util.Scanner class as follows:

```
Scanner input;
input = new Scanner(System.in);
int num = input.nextInt();
```

Standard Input

- The Scanner class acts as a wrapper for the input source
- It makes it easier to read basic data types from a character input source
- A scanner usually works with *tokens*

A Token:
A unit of meaningful string of characters that cannot be broken down into smaller meaningful pieces, they are separated by *whitespace*

Examples:

```
this line contains 5.0 tokens
1  67.3004  2  true                    // Four tokens
```

Common Scanner Methods

Method	Example
hasNext( )	true if stream has another token
hasNextLine( )	true if stream has at least one more line
hasNext*Type*( )	true if stream has token of *Type*
next( )	String str = scanner.next();
nextLine()	Reads a line as a value of type String
nextByte( )	byte b = scanner.nextByte( );
nextDouble( )	double d = scanner.nextDouble( );
nextFloat( )	float f = scanner.nextFloat( );
nextInt( )	int i = scanner.nextInt( );
nextLong( )	long l = scanner.nextLong( );
nextShort( )	short s = scanner.nextShort( );

Java File Input & Output

- Java input/output involving files and networks is based on streams
- Working with files and networks requires familiarity with exceptions
- Many of the methods used can throw exceptions that require mandatory exception handling
- Generally, this means calling the methods in a try..catch statement that can deal with the exception if one occurs

Java File Input & Output

- Human-readable character data is read from a file using an object of type FileReader, a subclass of Reader
- Data is written to a file in human-readable format through an object of type FileWriter, a subclass of Writer
- For files storing data in machine format, the appropriate I/O classes are FileInputStream and FileOutputStream
- Using FileReader and FileWriter classes is similar to using FileInputStream and FileOutputStream
- All these classes are defined in the java.io package.

Java File Input

- The FileReader class has a constructor which takes the name of a file as a parameter and creates an input stream that can be used for reading from that file
- When a FileReader is successfully created, we may need to wrap it in a Scanner class before we start reading data from it
- Example:

```
Scanner data;
// Declare the variable before the try statement, or else the variable
// is local to the try block and you won't be able to use it later in the program
try {
    data = new Scanner( new FileReader("data.dat") ); // create the stream
}
catch (FileNotFoundException e) {
    ... // do something to handle the error---maybe, end the program
}
```

Java File Output

- The FileWriter class works in a similar way to create an output stream of that type
- We may need to wrap this output stream in an object of type PrintWriter
- Example:

```
PrintWriter result;
try {
      result = new PrintWriter(new FileWriter("result.dat"));
}
catch (IOException e) {
      ... // handle the exception
}
```

Java File Output

- A file should be closed by calling the `close()` method of the associated stream
- Also, every output stream has a `flush()` method that can be called to force any data in the buffer to be written to the file without closing the file

Example1: Simple Java File I/O

```
import java.io.*;                       // Provides stream i/o classes
import java.util.*;                     // Provides ArrayList structure
/*
* Reads numbers from a file named data.dat and writes them to a file
* named result.dat in reverse order. The input file should contain
* exactly one real number per line.
*/
public class ReverseFile {
  public static void main(String[] args) {
    Scanner data;                       // Character input stream for reading data.
    PrintWriter result;                 // Character output stream for writing data.
    ArrayList<Double> numbers;          // An ArrayList for holding the data.
    numbers = new ArrayList<Double>();
```

Example1: Simple Java File I/O

```
try {                                   // Create the input stream.
    data = new Scanner(new FileReader("C:/data.dat")); }
catch (FileNotFoundException e) {
    System.out.println("Can't find file data.dat!");
    return;                             // End the program, returning from main.
}
try {                                   // Create the output stream.
    result = new PrintWriter(new FileWriter("C:/result.dat"));  }
catch (IOException e) {
    System.out.println("Can't open file result.dat!");
    System.out.println("Error: " + e);
    data.close();                       // Close the input file.
    return;                             // End the program.
   }
```

Example1: Simple Java File I/O

```
// Read numbers from the input file, adding them to the ArrayList.
    while ( data.hasNext()) {           // Read until end-of-file.
        double inputNumber = data.nextDouble();
        numbers.add( inputNumber );
    }
// Output the numbers in reverse order.
    for (int i = numbers.size()-1; i >= 0; i--)    result.println(numbers.get(i));
    System.out.println("Done!");
// Finish by closing the files, whatever else may have happened.
        data.close();
        result.close();
    }                                   // end of main
}                                       // end of class
```

- See the ReverseFile.java example

Example2: File I/O for Deep Copy

```
import java.io.Serializable;

// Implementing Serializable makes a class objects
// writable to binary streams.
public class Person implements Serializable {

        . . . . . .

}                                   // Class ends.
```

Example2: File I/O for Deep Copy

```
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;

//Define the filename to use
   String fileName = "C:/file.dat";

//Create the object that you want to deep copy later. Note that the object class
//must implement the Serializable interface
   Person[] p = new Person[2];          // Example structure that we want to copy
                .
                .
```

Example2: File I/O for Deep Copy

```
//Define the output stream by creating the space in memory for the output file
//stream  and connecting it to the file name specified above.
   ObjectOutputStream  os = null;

  try {
      os = new ObjectOutputStream(new FileOutputStream(fileName));
  }
  catch(IOException e) {
     System.out.println("Could not open the file." + e);
     System.exit(0);
  }
```

Example2: File I/O for Deep Copy

```
//Write the object to the binary file.
  try {
       os.writeObject(p);                  // Writes the array of the two person object
       os.close();                         // Flush output buffer and close the file
  }
  catch(IOException e) {
       System.out.println("Writing error: " + e);
       System.exit(0);
  }

//The write operation was successful.
  System.out.println("Records have been written to the file: " + fileName + ".");
  System.out.println();
```

Example2: File I/O for Deep Copy

```
//Define the input stream by creating the space in memory for the input file
//stream  and connecting it to the file name specified above.
   ObjectInputStream  is = null;

  try {
     is = new ObjectInputStream(new FileInputStream(fileName));
  }
  catch(IOException e) {
    System.out.println("There was a problem opening the file." + e);
    System.exit(0);
  }

//Create new blank object to store the incoming data.
  Person[] newp = null;                    // Example copy-destination structure.
```

Example2: File I/O for Deep Copy

```
//Read the object from the binary file.
  try {
    newp = (Person[]) is.readObject(); // Reads the array of the two person object
    is.close();                        // Close the input file
  }
  catch(IOException e) {
       System.out.println("There was an issue reading from the file: " + e);
       System.exit(0);
  }

//The Reading operation was successful.
  System.out.println("Reading of the file: " + fileName + has been completed.");
  System.out.println();
```

JAVA Language Review

Enumerations

The `Enum` Reference Type

- An enum is a type that has a fixed list of possible values, specified when it is created
- Syntax: (simplified form)

  enum *enum-type-name* { *list-of-enum-values* }

  Where:

  The *enum-type-name* is any identifier, will become the name of the enum type

  The *list-of-enum-values* is a list of identifiers (all Capitals) separated by commas
- This definition cannot be inside of a method
- It can be placed outside the main() method of the program
- Examples:

```
enum Season { SPRING, SUMMER, FALL, WINTER };
enum Action {START, STOP, REWIND, FORWARD};
enum BaseColor {RED, GREEN, BLUE};
```

The Enum Reference Type

- Enum values are not variables. Each value is a constant that always has the same value
- Example:

```
enum Season { SPRING, SUMMER, FALL, WINTER }
Season vacation;
vacation = Season.SUMMER;
System.out.print(vacation);              // Prints SUMMER
```

- An enum is a special type of a class and the enum values are objects that can have methods

The Enum Reference Type

- One of the methods in every enum value is named ordinal()
- It returns the ordinal number of the value in the list of values of the enum, which is the position of the value in the list
- Examples:

```
Season.SPRING.ordinal() is the int value 0,
Season.SUMMER.ordinal() is 1,
Season.FALL.ordinal() is 2, and
Season.WINTER.ordinal() is 3
```

- See the EnumDemo.java Example

JAVA Language Review

Object Oriented Design

Design Goals

- Robustness:
  - Correct programs that produce the right output for all input cases
  - Should handle all unexpected input cases without loosing its integrity
- Adaptability:
  - Ability of the program to run with minimal changes on different hardware and Operating System platforms
- Reusability:
  - The same code should be usable as a component of different systems in various applications

Object Oriented Design Principles

- Abstraction:
  - Being able to describe the most important parts of a system in simple and precise language, by naming them and explaining their functionality
  - Abstraction separates the definition (what) from the implementation (how)
  - In data structures, this is to describe ADTs
  - In Java, an ADT can be expressed by an *interface* (a list of method declarations)
  - In Java, an ADT can be implemented by a *class*
  - Thus, a java class is said to implement an interface if its methods include all methods declared in the interface

Object Oriented Design Principles

- Encapsulation:
  - Hiding of the details of the implementation, so it gives the programmer the freedom of how to implement a system
  - The programmer has to maintain the abstract interface that the users of the system see
- Modularity:
  - An organizing principle for code, in which different components of a software system are divided into separate functional units
  - Hierarchical organization of the modules helps to enable software reusability

Inheritance and Polymorphism

- The way Object Oriented design can provide reusability is through two powerful concepts:
  - Inheritance:
    - Allows designing general classes that can be specialized to more particular classes
    - The general class is known as the *superclass*
    - The new class is known as a *subclass*
    - The subclass can:
      - Reuse (inherit) the general methods defined in its superclass(es)
      - Re-implement or specialize (override) some of the general methods of its superclass(es)
      - Extend its superclass by implementing new methods

Inheritance and Polymorphism

- Polymorphism:
  - Allows many forms of a method to exist
    - By overriding methods from its superclass(es), or
    - By overloading methods of its class, so each one would have a different signature
  - Method Overriding Types:
    - Replacement: (All regular methods)
      The new method completely replaces the method of its superclass
    - Refinement: (for constructors – Constructor chaining)
      The new method adds to the method of its superclass:
      - Explicitly: By using the keyword super (first line)
      - Implicitly: Java will add a call to super()

Method Overriding Example

```
class A {                          // Define a class named A
  int i = 1;                       // An instance field
  int f() { return i; }            // An instance method
  static char g() { return 'A'; }  // A class method
}

class B extends A {                // Define a subclass of A
  int i = 2;                       // Hides field i in class A
  int f() { return -i; }           // Overrides instance method f in class A
  static char g() { return 'B'; }  // Hides class method g() in class A
}
```

Method Overriding Example

```
public class OverrideTest {
  public static void main(String args[ ]) {
    B b = new B();                 // Creates a new object of type B
    System.out.println(b.i);       // Refers to B.i; prints 2
    System.out.println(b.f());     // Refers to B.f(); prints -2
    System.out.println(b.g());     // Refers to B.g(); prints B
    System.out.println(B.g());     // This is a better way to invoke B.g()

    A a = (A) b;                   // Casts b to an instance of class A
    System.out.println(a.i);       // Now refers to A.i; prints 1
    System.out.println(a.f());     // Still refers to B.f(); prints -2
    System.out.println(a.g());     // Refers to A.g(); prints A
    System.out.println(A.g());     // This is a better way to invoke A.g()
  }
}
```

JAVA Language Review

Interfaces

Interfaces

- An interface is a collection of implicitly *abstract* method declarations with no data and no bodies
- It describes how a class interacts with its clients
- Syntax:

```
[modifiers] interface InterfaceName
        [extends Interface_NameList] {

        Member_Signatures;
}
```

Why an `interface` construct?

- Good software engineering
  - Specify and enforce boundaries between different parts of a team project
- You can use interface as a type
  - Allows more generic code
  - Reduces code duplication
- Unlike classes, types do not form a tree!
  - A class may implement several interfaces
  - An interface may be implemented by several classes

Notes:

- An interface is not a class!
  - It cannot be instantiated
  - It has incomplete specification
- Methods of an interface may not be declared *static*
- All members of an interface are implicitly *public*
- The only fields allowed in an interface definition are constants that are declared both *static* and *final*

Notes:

- When a class implements an interface it must implement all its methods
- Thus, an interface forces a class to implement methods of certain signatures
- Interfaces can be extended by other interfaces
- An interface in Java can inherit members from many other interfaces (multiple inheritance)
- A class in Java can inherit members from only a single superclass, but it can implement any number of interfaces
- Any instances of that class are members of both the type defined by the class and the type defined by the interface

Example:

```
/** Interface for objects that can be sold */
public interface Sellable {
    public String description();
    public int listPrice();
    public int lowestPrice();
}
/** Interface for objects that can be transported */
public interface Transportable {
    public int weight();
    public boolean isHazardous();
}
/** Interface for objects that can be insured */
public interface Insurable extends Transportable, Sellable {
    public int insuredValue();
}
```

Example:

```
/** Class for photographs that can be sold */
public class Photograph implements Sellable {
    private String descript;
    private int price;
    private boolean color;
    public Photograph(String desc, int p, boolean c) {
        descript = desc;
        price = p;
        color = c;
    }
    public String description() { return descript; }
    public int listPrice() { return price; }
    public int lowestPrice() { return price/2; }
    public boolean isColor() { return color; }
}
```

Example:

```
/** Class for objects that can be sold, packed, insured, and shipped */
public class BoxedItem implements Insurable {
    private String description;
    private int price;
    private int weight;
    private boolean haz;
    private int height=0;
    private int width=0;
    private int depth=0;

    public BoxedItem(String desc, int p, int w, boolean h) {
            description = desc;
            price = p;
            weight = w;
            haz = h;
    }
```

Example:

```
    public String description() { return description; }
    public int listPrice() { return price; }
    public int lowestPrice() { return price/2; }
    public int weight() { return weight; }
    public boolean isHazardous() { return haz; }
    public int insuredValue() { return price*2; }
    public void setBox(int h, int w, int d) {
        height = h;
        width = w;
        depth = d;
    }
}
```

Empty Slide

JAVA Language Review

Generics

Generic Types

- Definition:
  - Similar to defining a function with a set of formal variable parameters, a generic type allows the definition of a class in terms of a set of formal type parameters
  - A generic type is defined using one or more type variables and has one or more methods that use a type variable as a placeholder for an argument or return type
- Purpose:
  - Generic programming produces highly general, reusable and compile-time checked type-safe code
  - Generic types are not defined at compilation time, but they become fully specified at run time

1. Simple Generic Classes

- Syntax:
  a. [**public**] **class** *ClassName* [< *generic-param* [, *generic-param*]... >] { *body* }

  b. [**public**] **interface** *InterfaceName* [< *generic-param* [, *generic-param*]... >] { *body* }

  - Where, *generic-param* is any name. Java style, one letter: **E**, **T**, **S** or **V**
  - The *generic-param* in the definition will be replaced by an actual type name when the class is used to declare variables or create objects
- The actual parameter replacing the *generic-param* can be any type
- In this kind of generics, the type parameters are added to the names of classes and interfaces only, never to the names of methods or constructors

1. Simple Generic Classes

- Notes:
  - Class generic type names can be used anywhere a type is required in any instance fields or methods of the class
  - Class generic type names are not allowed in static fields or methods of the class
  - Creating new arrays of generic type is not allowed because it is not type-safe
  - The generic type names exist only at compile time, they can't be used with the runtime operators instanceof and new

1. Simple Generic Classes
Example 1: The Pair Class

```
/** A class of pair that contains two objects, possibly of different types */
public class Pair<T,S> {                        // T and S are type parameters
    public T first;                             // Instance variable of type T
    public S second;                            // Instance variable of type S
    public Pair( T a, S b ) { first = a; second = b; }      // Constructor
    public T getFirst( ) { return first; }                  // Methods
    public S getSecond( ) { return second; }
    public String toString( ) { return "[" + getFirst() + ", " + getSecond() + "]"; }

    public static void main (String[ ] args) {
        Pair<String,Integer> pair1 = new Pair<String,Integer>("Height", 157);
        System.out.println (pair1);

        Pair<BoxedItem,Double> pair2 =
            new Pair<BoxedItem,Double>(new BoxedItem(…), new Double(42.8));
        System.out.println (pair2);
}                                               // See Pair.java program
```

1. Simple Generic Classes
Example 2: The Tree Class

```
import java.util.*;

// A tree is a data structure that holds values of type V. Each tree has a single
// value of type V and can have any number of branches, each is itself a Tree.
public class Tree<V> {
    V value;
    List<Tree<V>> branches = new ArrayList<Tree<V>>();
    public Tree(V value) { this.value = value; }            // The constructor

    // Instance methods for manipulating the node value and branches.
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<V> branch) { branches.add(branch); }
}
```

2. Simple Generic Methods

- Static methods cannot use the type variables of their containing class
- But they can declare their own type variables in their signature, before specifying their return type
- Such methods are called generic methods
- Syntax:

```
public static [modifiers] [< generic-param(s) >]
type methodName ( paramlist ) [ throws exceptions ] {  MethodBody }
```

- All methods, including static methods, can declare and use their own type parameters
- Each invocation of such a method can be parameterized differently

2. Simple Generic Methods
Generic Definition Example

- Generic methods are required where a single type variable is used to express a relationship between two parameters or between a parameter and a return value
- Example:

```
// Returns the number of times that itemToCount occurs in the list.
// Items in the list are tested for equality using the equals() method,
// except in the special case where itemToCount is null.
public static <T> int countOccurrences(T[ ] list, T itemToCount) {
    int count = 0;
    if (itemToCount == null) {
        for ( T listItem : list ) if (listItem == null) count++;
    } else {
        for ( T listItem : list ) if (itemToCount.equals(listItem)) count++;
    }
    return count;
}
```

2. Simple Generic Methods
Generic Method Invocation

1. If wordList is a variable of type String[ ] and word is a variable of type String, then we count the number of times word occurs in wordList by:

```
int count = countOccurrences( wordList, word );
```

2. If palette is a variable of type Color[ ] and color is a variable of type Color, then we count the number of times color occurs in palette by:

```
int count = countOccurrences( palette, color );
```

3. If numbers is a variable of type Integer[ ], then we count the number of times that 17 occurs in numbers by:

```
int count = countOccurrences( numbers, 17 );
```

2. Simple Generic Methods
Generic Method Invocation

- Note that when a generic method is used, there is no need to explicitly mention the type to be substituted for the type parameter
- The compiler, in this case, deduces the type from the types of the actual parameters in the method call

Problem with Simple Generics

- Problem:

  Since the type parameter named T, can be any type at all, it means that its usage is restricted to operations that can be done with Objects only

- Examples:
  - It is not possible to write a generic method that compares objects with the compareTo() method, because it is not defined for all objects. It is defined in the Comparable interface
  - It is not possible also to write generic classes like List<Object> and List<Integer> that are type-safe, because a String is an object but can not be cast to an Integer which is not assignment compatible with it

3. Wildcard Generics

Wildcard Generics:

A wildcard type is used as type parameter in declaring variables and formal method parameters

Syntax:

generic-param is written as: **?** [extends *type* | super *type*]

Notes:

- The extends clause defines an *upper type bound* for the wildcard type **?**
- The super clause defines a *lower type bound* for the wildcard type **?**
- Wildcards are used to generalize method definitions, so that they can work with collections of objects of various unknown types, rather than just a single type

3. Wildcard Generics

Example 1:

```
// a method to display the elements of a List
public static void printList(PrintWriter out, List<?> list) {
    for (int i=0, n=list.size(); i < n; i++) {
        if (i > 0) out.print(", ");
        Object o = list.get(i);
        out.print(o.toString());
    }
}
```

Notes:

- Use a ? wildcard if a type is generic and the value of the type variable is unknown
- Wildcard can not be used when invoking a constructor

3. Wildcard Generics
Example 2: The Tree Class

```
import java.util.*;

// A tree is a data structure that holds values of type V. Each tree has a single
// value of type V and can have any number of branches, each is itself a Tree.
public class Tree<V> {
    V value;
    List<Tree<? extends V>> branches = new ArrayList<Tree<? extends V>>();
    public Tree(V value) { this.value = value; }              // The constructor

    // Instance methods for manipulating the node value and branches.
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<? extends V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<? extends V> branch) { branches.add(branch); }
}
```

4. Bounded Generics

Bounded (Restricted) Generics:

To enforce that:

- A type parameter implements one or more interfaces, or
- A type parameter is a subclass of a specified class

Bounded types are used to restrict the allowed formal type parameters in a generic class or method or interface definition. Wildcards can not be used to do that

Syntax:

generic-param is written as: *name* [extends *type*]

4. Bounded Generic Example
Bounded Generic Method

- Example:

```
// This method returns the largest of two trees of generic bounded
// Number type, where tree size is computed by the sum() method.
// The type variable ensures that both trees have the same value
// type and that both can be passed to sum().
public static <N extends Number>
Tree<N> max(Tree<N> t, Tree<N> u) {
    double ts = sum(t);
    double us = sum(u);
    if (ts > us) return t;
    else return u;
}
```

4. Bounded Generic Example Bounded (Generic vs. Wildcard)

- Example: (Restricted generic method)

```
// Recursively compute the sum of the values of all nodes on the tree
public static <N extends Number> double sum(Tree<N> t) {
    N value = t.value;
    double total = value.doubleValue();
    for (Tree<? extends N> b : t.branches) total += sum(b);
    return total;
}
```

- Example: (Wildcard method – Preferred)

```
// Recursively compute the sum of the values of all nodes on the tree
public static double sum(Tree<? extends Number> t) {
    double total = t.value.doubleValue();
    for (Tree<? extends Number> b : t.branches) total += sum(b);
    return total;
}
```

4. Bounded Generics

Notes:

- A type variable can have any number of bounds, including any number of interfaces and at most one class
- Example:

```
public class Tree<V extends Serializable & Comparable<V>>
                implements Serializable, Comparable<Tree<V>> {
```

- The extends clause defines an upper type bound for the type parameter *name*