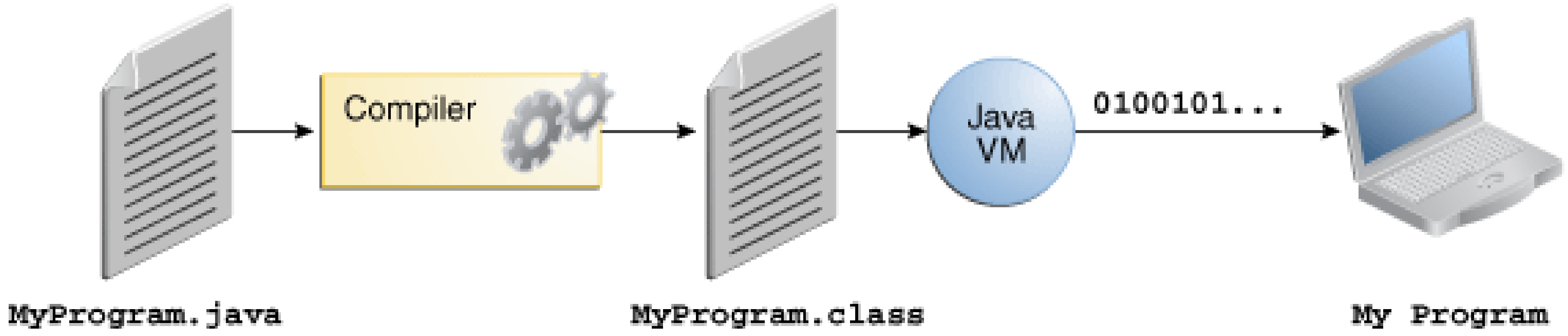
Java Review

ACS-2947 LECTURE 0
(Chapter 1)

The Java programming language

- All source code is first written in plain text files ending with the .java extension
- Those source files are then compiled into .class files by the javac compiler.
- A .class file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the Java Virtual Machine (Java VM)
- The java launcher tool then runs your application with an instance of the Java Virtual Machine.


Compiler
Java VM
0100101...
MyProgram.java
MyProgram.class
My Program

Hello World!

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

Comments

```
// This is an inline comment

/*
* This is a block comment
*/
```

Base types

- For the most commonly used data types, Java provides the following base types or primitive data types:

boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)

Base types (Examples)

```
boolean flag = true;
boolean verbose, debug;
char grade = 'A';
byte b = 12;
short s = 24;
int i, j, k = 257;
long l = 890L;
float pi = 3.1416F;
double e = 2.71828, a = 6.022e23;
```

Classes and Objects

- Class
  - the type of the object
  - blueprint
    - defines the data which the object stores and the methods for accessing and modifying that data
- Object
  - Instance of a class

Class members

- The critical members of a Java class
  - Instance variables (fields)
    - Represent the data associated with an object
    - Must have a type (base or class)
  - Methods
    - Code that can be called to perform actions
      - Accessor method: returns information without changing instance variables
      - Update method: may change one or more instance variables when called

```
public class Counter {
  private int count;                                        // a simple integer instance variable
  public Counter( ) { }                                     // default constructor (count is 0)
  public Counter(int initial) { count = initial; }                   // an alternate constructor
  public int getCount( ) { return count; }                               // an accessor method
  public void increment( ) { count++; }                                   // an update method
  public void increment(int delta) { count += delta; }                    // an update method
  public void reset( ) { count = 0; }                                     // an update method
}
```

- This class includes
  - one instance variable (count), which will have a default value of zero, unless we otherwise initialize it.
  - two special methods known as constructors, one accessor method, and three update methods.

Reference variables

- Classes are known as ***reference types*** in Java, and a variable of that type is known as a **reference variable**.
  - capable of storing the location (i.e., memory address) of an object from the declared class.
  - can reference an existing instance or a newly constructed instance.
  - can also store a special value, null, that represents the lack of an object.

Defining classes

- Access control modifiers
  - **public** designates that all classes may access the defined aspect.
  - **protected** designates that access is only granted to classes that are designated as subclasses of the given class through inheritance or in the same package.
  - **private** designates that access to a defined member of a class be granted only to code within that class.

The static modifier

- The **static** modifier in Java can be declared for any variable or method of a class. It is associated with the class itself, and not with any particular instance of that class
  - A static variable
    - used to store “global” information about a class, exist even if no instance of their class exists.

The static modifier

- A static method
  - not invoked on a particular instance of the class using the traditional dot notation, it is typically invoked using the name of the class as a qualifier.
  - can be useful for providing utility behaviors related to a class that need not rely on the state of any particular instance of that class.
  - E.g. Math.sqrt(2)

The abstract modifier

- A method of a class may be declared as **abstract** in which its signature is provided but without an implementation of the method body.
  - any subclass of a class with abstract methods is expected to provide a concrete implementation for each abstract method.
- Abstract methods are an advanced feature of object-oriented programming to be combined with inheritance
  - More on this later

The final modifier

- A variable that is declared with the final modifier can be initialized as part of that declaration, but can never again be assigned a new value.
  - If it is a base type, then it is a constant. If a reference variable is final, then it will always refer to the same object
  - If a member variable of a class is declared as final, it will typically be declared as static as well

Methods

- 2 parts:
  - The signature defines name and parameters
  - The body defines what the method does

[*modifiers*] *returnType methodName*(*type*$_1$ *param*$_1$, …, *type*$_n$ *param*$_n$) {
    // method body
}

```
public void increment (int delta){
        count += delta;
}
```

Parameters

- A method’s parameters are defined in a comma-separated list enclosed in parentheses after the name of the method.
  - A parameter consists of two parts, the parameter type and the parameter name.
  - If a method has no parameters, then only an empty pair of parentheses is used.

Parameters

- All parameters in Java are **passed by value**, that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body.
  - So if we pass an int variable to a method, then that variable’s integer value is copied.
  - The method can change the copy but not the original.

Signatures

- If there are several methods with this same name defined for a class, then the Java runtime system uses the one that matches the actual number of parameters sent as arguments, as well as their respective types.
- A method’s name combined with the number and types of its parameters is called a method’s ***signature***, for it takes all of these parts to determine the actual method to perform for a certain method call.

Recall:

```
public class Counter {
  private int count;                                            // a simple integer instance variable
  public Counter( ) { }                                         // default constructor (count is 0)
  public Counter(int initial) { count = initial; }                   // an alternate constructor
  public int getCount( ) { return count; }                               // an accessor method
  public void increment( ) { count++; }                                   // an update method
  public void increment(int delta) { count += delta; }                    // an update method
  public void reset( ) { count = 0; }                                     // an update method
}
```

```
 1  public class CounterDemo {
 2    public static void main(String[ ] args) {
 3      Counter c;                      // declares a variable; no counter yet constructed
 4      c = new Counter( );             // constructs a counter; assigns its reference to c
 5      c.increment( );                 // increases its value by one
 6      c.increment(3);                 // increases its value by three more
 7      int temp = c.getCount( );       // will be 4
 8      c.reset( );                     // value becomes 0
 9      Counter d = new Counter(5);// declares and constructs a counter having value 5
10      d.increment( );                 // value becomes 6
11      Counter e = d;                  // assigns e to reference the same object as d
12      temp = e.getCount( );           // will be 6 (as e and d reference the same counter)
13      e.increment(2);                 // value of e (also known as d) becomes 8
14    }
15  }
```

- Here, a new Counter is constructed at line 4, with its reference assigned to the variable c. That relies on a form of the constructor, Counter( ), that takes no arguments between the parentheses.

The dot operator

- One of the primary uses of an object reference variable is to access the members of the class for this object, an instance of its class.
- This access is performed with the dot (".") operator.
- We call a method associated with an object by using the reference variable name, following that by the dot operator and then the method name and its parameters.

Constructors

- A **constructor** is a special kind of method that is used to initialize a newly created instance of the class so that it will be in a consistent and stable initial state.
  - Typically achieved by initializing each instance variable of the object (unless the default value will suffice)
  - A new object is created by using the **new** operator followed by a call to a constructor for the desired class.
  - It is a method that always shares the same name as its class. The new operator returns a reference to the newly created instance

Defining constructors

Constructors are defined in a very similar way as other methods of a class, but there are a few important distinctions:

1. Constructors cannot be static, abstract, or final, so the only modifiers that are allowed are those that affect visibility (public, protected, private)
2. The name of the constructor must be identical to the name of the class it constructs.
3. We don’t specify a return type for a constructor (not even void). Nor does the body of a constructor explicitly return anything.

Defining constructors

$modifiers\ name(type_1\ param_1,\ \ldots,\ type_n\ param_n)\ \{$

```
                    // constructor body
}
```

```
Counter d = new Counter(5);
```

- the new operator is responsible for returning a reference to the new instance to the caller;
- the responsibility of the constructor method is only to initialize the state of the new instance.

Defining constructors

- A class can have many constructors, but each must have a different ***signature***
    - each must be distinguished by the type and number of the parameters it takes.
- If no constructors are explicitly defined, Java provides an implicit default constructor for the class, having zero arguments and leaving all instance variables initialized to their ***default values***.
    - If a class defines one or more nondefault constructors, no default constructor will be provided.

The keyword **this**

Within the body of a method in Java, the keyword **this** is automatically defined as a reference to the instance upon which the method was invoked. There are three common uses:

1. To store the reference in a variable, or send it as a parameter to another method that expects an instance of that type as an argument.
2. To differentiate between an instance variable and a local variable with the same name.
3. To allow one constructor body to invoke another constructor body.

The main method

- Some Java classes are meant to be used by other classes, but are not intended to serve as a self-standing program.
- The primary control for an application in Java must begin in some class with the execution of a special method named main.
- This method must be declared as follows:

```
public static void main(String[ ] args) {
  // main method body...
}
```

- Strings
- Wrappers
- Arrays
- Enum Types

The String class

- Because it is common to work with sequences of text characters in programs, Java provides support in the form of a String ***class***

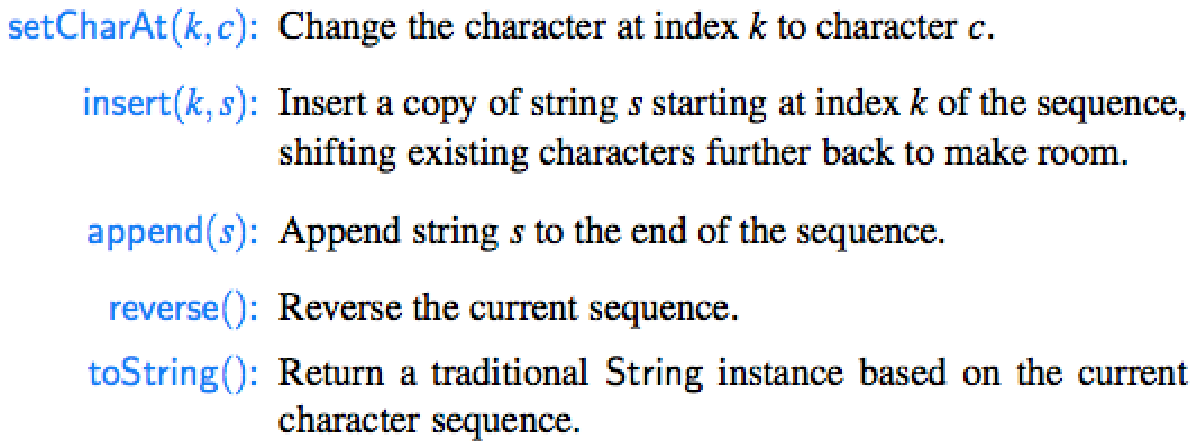
```
String title = "Data Structures & Algorithms in Java";
```

  - Each character c within a string s can be referenced by using an ***index***, which is equal to the number of characters that come before c in s.
  - In Java, the “+” operation performs concatenation when acting on two strings, as follows:

```
String term = "over" + "load";
```

The StringBuilder class

- In order to support more efficient editing of character strings, Java provides a StringBuilder class, which is effectively a ***mutable*** version of a string.
  - combines some of the accessor methods of the String class, while supporting additional methods



setCharAt(k,c):	Change the character at index k to character c.
insert(k, s):	Insert a copy of string s starting at index k of the sequence, shifting existing characters further back to make room.
append(s):	Append string s to the end of the sequence.
reverse():	Reverse the current sequence.
toString():	Return a traditional String instance based on the current character sequence.

Wrapper types

- There are many data structures and algorithms in Java’s libraries that are specifically designed so that they only work with object types (not primitives).
  - To get around this obstacle, Java defines a wrapper class for each base type. An instance of each wrapper type stores a single value of the corresponding base type.

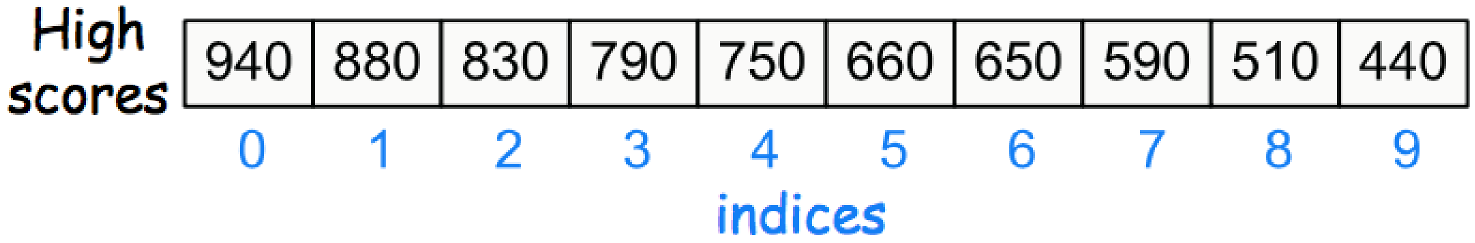
Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

Automatic boxing and unboxing

- Java provides additional support for implicitly converting between base types and their wrapper types
  - In any context for which an Integer is expected (for example, as a parameter), an int value k can be expressed, in which case Java automatically ***boxes*** the int, with an implicit call to new Integer(k)
  - In reverse, in any context for which an int is expected, an Integer value v can be given in which case Java automatically ***unboxes*** it with an implicit call to v.intValue( )

Arrays

- A common programming task is to keep track of an ordered sequence of related values or objects
- An array is a sequenced collection of variables of all the same type.
  - Each variable (cell) in an array has an index, which uniquely refers to the value stored in that cell



Arrays

- Each value stored is called an ***element*** of that array
    - a[k]
- The length of an array is referred to as its ***capacity***
    - a.length
- An attempt to index an array a using a number outside of the range from 0 to a.length-1 results in an ***out of bounds*** reference
    - ArrayIndexOutOfBoundsException

Declaring and constructing arrays

- To declare, use an empty pair of square brackets
  - E.g. **int**[ ] primes;
- 2 ways to construct an array
  - Assignment to a literal form
    - *elementType*[ ] *arrayName* = {$initialValue_0$, $initialValue_1$, . . . , $initialValue_{N-1}$};
    - E.g.

```
int[ ] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

Declaring and constructing arrays

- Use the **new** operator
  - **new** *elementType*[*length*]
  - E.g.

```
double[ ] measurements = new double[1000];
```

Enum types

- Java supports an approach to representing choices from a finite set by defining what is known as an enumerated type, or enum for short.
    - These are types that are only allowed to take on values that come from a specified set of names.

Enum types

- For example, an enumerated type definition for days of the week might appear as:

```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
```

- Once defined, Day becomes an official type and we may declare variables or parameters with type Day. A variable of that type can be declared as: `Day today;`
- and an assignment of a value to that variable can appear as:

```
today = Day.TUE;
```

EXPRESSIONS

- Literals
- Operators
- Type conversions

Literals

- A ***literal*** is any “constant” value that can be used in an assignment or other expression.
  - Java allows the following kinds of literals:
    - Boolean
    - Integer
    - Floating point
    - Character
    - String
    - null object reference

Operators

- Java supports the following arithmetic operators:

+	addition
−	subtraction
*	multiplication
/	division
%	the modulo operator

  - If both operands have type int, then the result is an int; if one or both operands have type float, the result is a float.
  - Integer division has its result truncated.

Increment and decrement operators

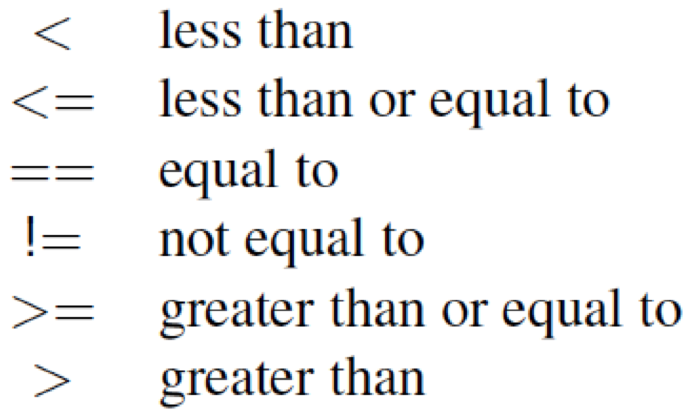
- Java provides the increment (++) and decrement (−−) operators.
  - If such an operator is used in front of a variable reference, then 1 is added to (or subtracted from) the variable and its value is read into the expression.
  - If used after a variable reference, then the value is first read and then the variable is incremented or decremented by 1.

Increment and decrement operators

```
int i = 8;
int j = i++;          // j becomes 8 and then i becomes 9
int k = ++i;          // i becomes 10 and then k becomes 10
int m = i--;          // m becomes 10 and then i becomes 9
int n = 9 + --i;      // i becomes 8 and then n becomes 17
```

Logical operators

- Java supports the following operators for numerical values, which result in Boolean values:



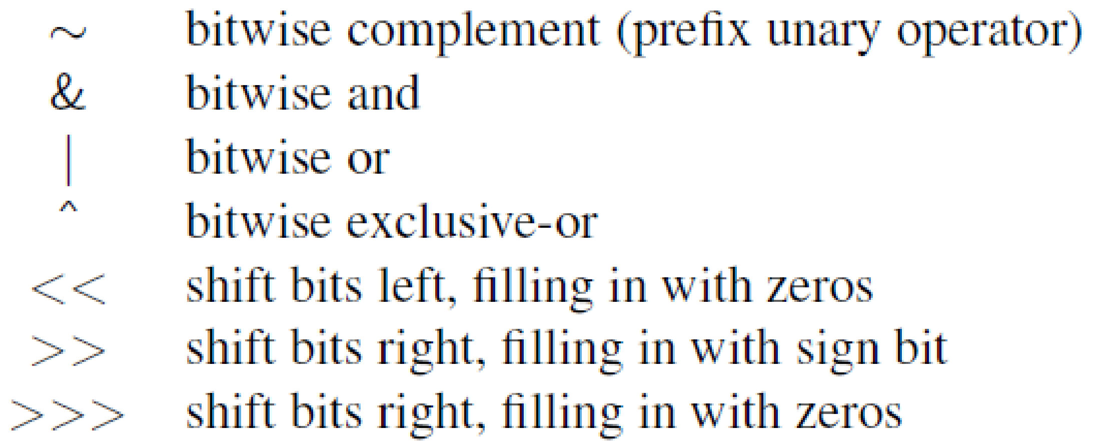
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

Logical operators

- Boolean values also have the following operators:

!	not (prefix)
&&	conditional and
||	conditional or

Bitwise operators



~	bitwise complement (prefix unary operator)
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<<	shift bits left, filling in with zeros
>>	shift bits right, filling in with sign bit
>>>	shift bits right, filling in with zeros

Assignment operator

The standard assignment operator in Java is “=”. It is used to assign a value to an instance variable or local variable.

variable = expression

E.g. x = 3

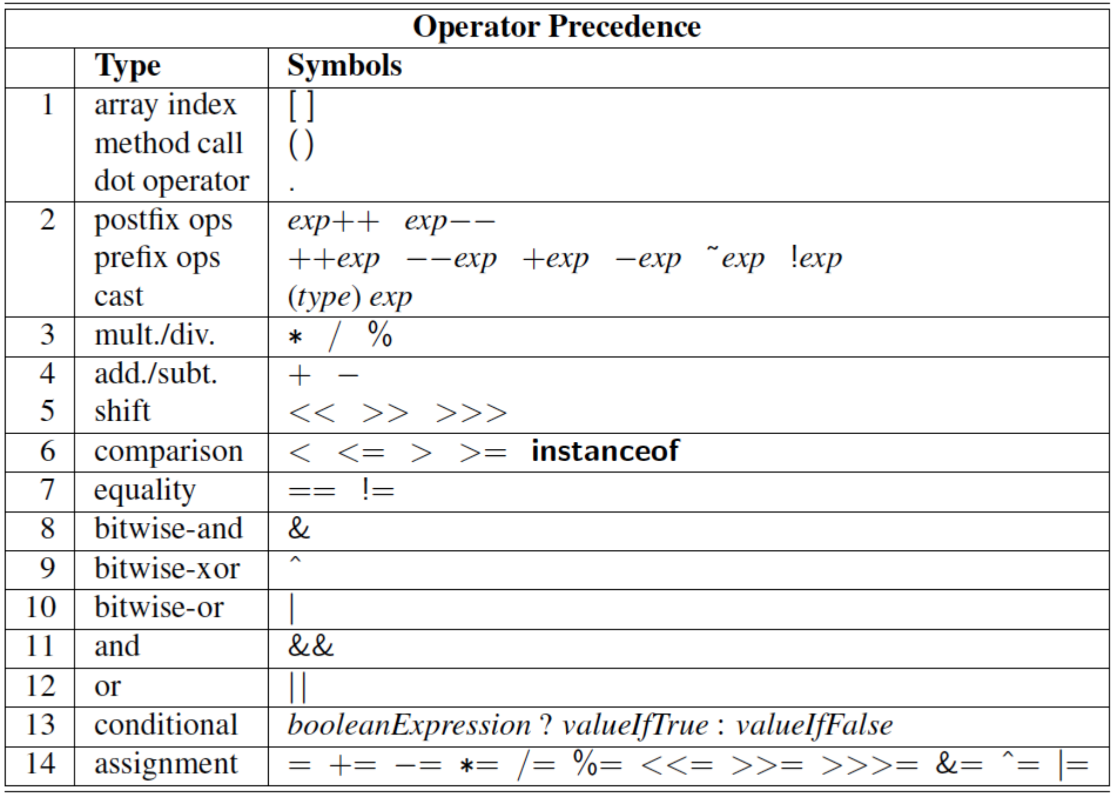
Compound assignment operator

variable op= expression which is generally equivalent to *variable = variable op expression*

E.g.

```
x += 2
x = x + 2
```

Operator precedence



Operator Precedence		
	Type	**Symbols**
1	array index	[ ]
	method call	( )
	dot operator	.
2	postfix ops	*exp*++ *exp*−−
	prefix ops	++*exp* −−*exp* +*exp* −*exp* ˜*exp* !*exp*
	cast	(*type*) *exp*
3	mult./div.	∗ / %
4	add./subt.	+ −
5	shift	<< >> >>>
6	comparison	< <= > >= **instanceof**
7	equality	== !=
8	bitwise-and	&
9	bitwise-xor	ˆ
10	bitwise-or	|
11	and	&&
12	or	||
13	conditional	*booleanExpression* ? *valueIfTrue* : *valueIfFalse*
14	assignment	= += −= ∗= /= %= <<= >>= >>>= &= ˆ= |=

Type conversions

- ***Casting*** is an operation that allows us to change the type of a value.
- We can take a value of one type and cast it into an equivalent value of another type.
- There are two forms of casting in Java: **explicit** casting and **implicit** casting

Explicit casting

- Java supports an explicit casting syntax with the following form:

  (*type*) *exp*

  - Here "*type*" is the type that we would like the expression *exp* to have.
  - This syntax may only be used to cast from one primitive type to another primitive type, or from one reference type to another reference type.

Explicit casting

- ***widening*** cast E.g. from an int to a double
    - as the double type is more broad than the int type, and a conversion can be performed without losing information.
- ***narrowing*** cast E.g. from a double to an int is
    - we may lose precision, as any fractional portion of the value will be truncated.

Explicit casting

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1;                          // i1 gets value 3
int i2 = (int) d2;                          // i2 gets value 3
double d3 = (double) i2;                    // d3 gets value 3.0
```

Implicit Casting

- There are cases where Java will perform an ***implicit cast*** based upon the context of an expression.
- You can perform a widening cast between primitive types (such as from an int to a double), without explicit use of the casting operator.
- However, if attempting to do an implicit narrowing cast, a compiler error results.

```
int i1 = 42;
double d1 = i1;                  // d1 gets value 42.0
i1 = d1;                         // compile error: possible loss of precision
```

Control Flow

- Conditionals
  - If
  - Switch
- Loops
  - While
  - Do-while
  - For
  - For-each

If statements

- The syntax of a simple if statement is as follows:

```
if  (booleanExpression)
        trueBody
else
        falseBody
```

  - *booleanExpression* is a boolean expression and *trueBody* and *falseBody* are each either a single statement or a block of statements enclosed in braces (“{” and “}”).

Compound if statements

- There is also a way to group a number of boolean tests, as follows:

```
if (firstBooleanExpression)
    firstBody
else if (secondBooleanExpression)
    secondBody
else
    thirdBody
```

Switch statements

- Java provides for multiple switch statement.

```
switch (d) {
  case MON:
    System.out.println("This is tough.");
    break;
  case TUE:
    System.out.println("This is getting better.");
    break;
  case WED:
    System.out.println("Half way there.");
    break;
  case THU:
    System.out.println("I can see the light.");
    break;
  case FRI:
    System.out.println("Now we are talking.");
    break;
  default:
    System.out.println("Day off!");
}
```

While loops

- The simplest kind of loop in Java is a while loop.
- Such a loop tests that a certain condition is satisfied and will perform the body of the loop each time this condition is evaluated to be true.

```
while (booleanExpression)
        loopBody
```

```
int j = 0;
while ((j < data.length) && (data[j] != target))
  j++;
```

Do-while loops

- Java has another form of the while loop that allows the boolean condition to be checked at the end of each pass of the loop rather than before each pass.

```
do
        loopBody
while (booleanExpression)
```

```
String input;
do {
  input = getInputString( );
  handleInput(input);
} while (input.length( ) > 0);
```

For loops

- The traditional **for**-loop syntax consists of four sections—an initialization, a boolean condition, an increment statement, and the body—although any of those can be empty.
- The structure is as follows:

```
for (initialization;
    booleanCondition; increment)
                loopBody
```

```
{
    initialization;
    while (booleanCondition) {
        loopBody;
        increment;
    }
}
```

For-each loops

- Since looping through elements of a collection is such a common construct, Java provides a shorthand notation for such loops, called the for-each loop.

for (*elementType name* : *container*)

loopBody

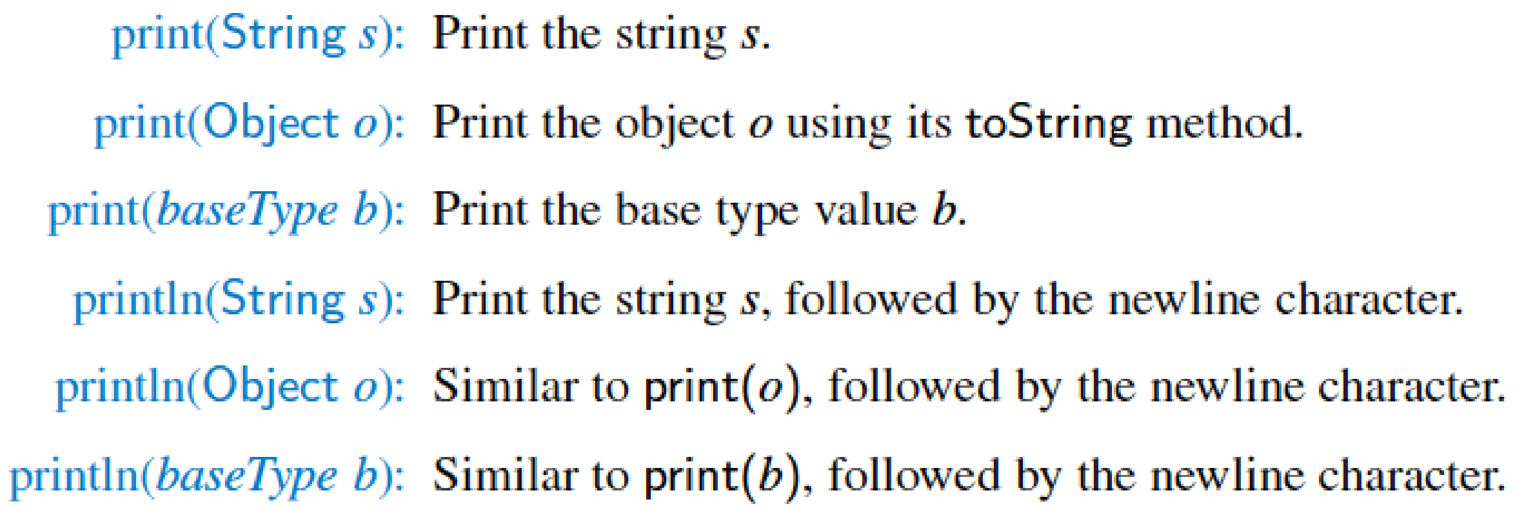
```
public static double sum(double[ ] data) {
  double total = 0;
  for (double val : data)                           // Java's for-each loop style
    total += val;
  return total;
}
```

Explicit control-flow statements

- The **return** statement exits a method and the flow of control returns
    - With argument (with appropriate value) or without (return type void)
- The **break** statement can be used to “break” out of the innermost switch, for, while, or do-while statement body.
- The **continue** statement causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

Simple output

- Java provides a built-in static object, called System.out, that performs output to the "standard output" device, with the following methods:



print(String *s*):	Print the string *s*.
print(Object *o*):	Print the object *o* using its toString method.
print(*baseType b*):	Print the base type value *b*.
println(String *s*):	Print the string *s*, followed by the newline character.
println(Object *o*):	Similar to print(*o*), followed by the newline character.
println(*baseType b*):	Similar to print(*b*), followed by the newline character.

Simple input

- System.in performs input from the Java console window.
- A simple way of reading input with this object is to use it to create a Scanner object, using the expression:
  - **new** Scanner(System.in)

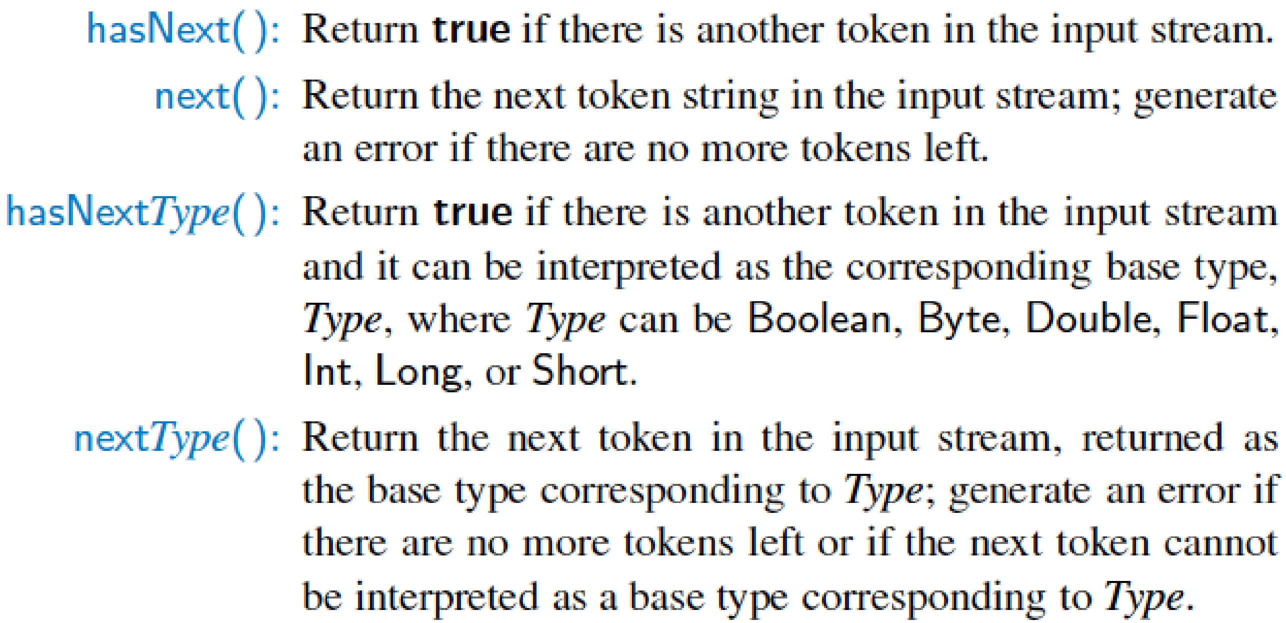
System input

```
import java.util.Scanner;                    // loads Scanner definition for our use

public class InputExample {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your age in years: ");
    double age = input.nextDouble( );
    System.out.print("Enter your maximum heart rate: ");
    double rate = input.nextDouble( );
    double fb = (rate − age) ∗ 0.65;
    System.out.println("Your ideal fat-burning heart rate is " + fb);
  }
}
```

java.util.Scanner methods

- The Scanner class reads the input stream and divides it into tokens, which are strings of characters separated by delimiters.



hasNext( ):	Return **true** if there is another token in the input stream.
next( ):	Return the next token string in the input stream; generate an error if there are no more tokens left.
hasNext*Type*( ):	Return **true** if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be Boolean, Byte, Double, Float, Int, Long, or Short.
next*Type*( ):	Return the next token in the input stream, returned as the base type corresponding to *Type*; generate an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*.

```
public class CreditCard {
  // Instance variables:
  private String customer;        // name of the customer (e.g., "John Bowman")
  private String bank;            // name of the bank (e.g., "California Savings")
  private String account;         // account identifier (e.g., "5391 0375 9387 5309")
  private int limit;              // credit limit (measured in dollars)
  protected double balance;       // current balance (measured in dollars)
  // Constructors:
  public CreditCard(String cust, String bk, String acnt, int lim, double initialBal) {
    customer = cust;
    bank = bk;
    account = acnt;
    limit = lim;
    balance = initialBal;
  }
  public CreditCard(String cust, String bk, String acnt, int lim) {
    this(cust, bk, acnt, lim, 0.0);                    // use a balance of zero as default
  }
```

```
// Accessor methods:
public String getCustomer() { return customer; }
public String getBank() { return bank; }
public String getAccount() { return account; }
public int getLimit() { return limit; }
public double getBalance() { return balance; }
// Update methods:
public boolean charge(double price) {          // make a charge
  if (price + balance > limit)                 // if charge would surpass limit
    return false;                              // refuse the charge
  // at this point, the charge is successful
  balance += price;                            // update the balance
  return true;                                 // announce the good news
}
public void makePayment(double amount) {       // make a payment
  balance -= amount;
}
```

```
  // Utility method to print a card's information
  public static void printSummary(CreditCard card) {
    System.out.println("Customer = " + card.customer);
    System.out.println("Bank = " + card.bank);
    System.out.println("Account = " + card.account);
    System.out.println("Balance = " + card.balance);   // implicit cast
    System.out.println("Limit = " + card.limit);        // implicit cast
  }
  // main method shown on next page...
}
```

```java
public static void main(String[ ] args) {
  CreditCard[ ] wallet = new CreditCard[3];
  wallet[0] = new CreditCard("John Bowman", "California Savings",
                             "5391 0375 9387 5309", 5000);
  wallet[1] = new CreditCard("John Bowman", "California Federal",
                             "3485 0399 3395 1954", 3500);
  wallet[2] = new CreditCard("John Bowman", "California Finance",
                             "5391 0375 9387 5309", 2500, 300);

  for (int val = 1; val <= 16; val++) {
    wallet[0].charge(3*val);
    wallet[1].charge(2*val);
    wallet[2].charge(val);
  }

  for (CreditCard card : wallet) {
    CreditCard.printSummary(card);        // calling static method
    while (card.getBalance( ) > 200.0) {
      card.makePayment(200);
      System.out.println("New balance = " + card.getBalance( ));
    }
  }
}
```

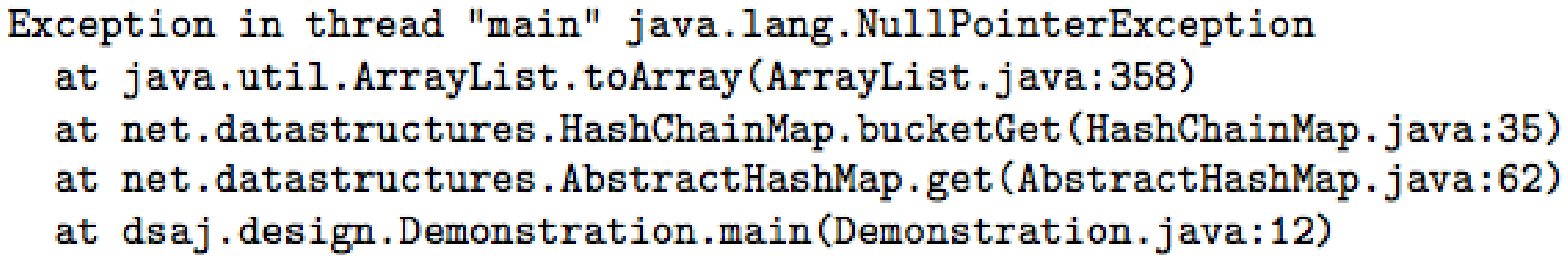
```
Customer = John Bowman
Bank = California Savings
Account = 5391 0375 9387 5309
Balance = 408.0
Limit = 5000
New balance = 208.0
New balance = 8.0
Customer = John Bowman
Bank = California Federal
Account = 3485 0399 3395 1954
Balance = 272.0
Limit = 3500
New balance = 72.0
Customer = John Bowman
Bank = California Finance
Account = 5391 0375 9387 5309
Balance = 436.0
Limit = 2500
New balance = 236.0
New balance = 36.0
```

Exceptions

- Exceptions are unexpected events that occur during the execution of a program.
- An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer.
- In Java, exceptions are objects that can be thrown by code that encounters an unexpected situation.
- An exception may also be caught by a surrounding block of code that “handles” the problem.

Catching exceptions

- If uncaught, an ***exception*** causes the virtual machine to stop executing the program and to report an appropriate message to the console.



```
Exception in thread "main" java.lang.NullPointerException
  at java.util.ArrayList.toArray(ArrayList.java:358)
  at net.datastructures.HashChainMap.bucketGet(HashChainMap.java:35)
  at net.datastructures.AbstractHashMap.get(AbstractHashMap.java:62)
  at dsaj.design.Demonstration.main(Demonstration.java:12)
```

- Before a program is terminated, each method on the stack trace has an opportunity to catch the exception.

Catching an exception

- The general methodology for handling exceptions is a try-catch construct in which a guarded fragment of code that might throw an exception is executed.

```
try {
    guardedBody
} catch ($exceptionType_1$  $variable_1$) {
    $remedyBody_1$
} catch ($exceptionType_2$  $variable_2$) {
    $remedyBody_2$
} ...
    ...
```

- If it throws an exception, then that exception is caught by having the flow of control jump to a predefined catch block that contains the code to apply an appropriate resolution.
- If no exception occurs in the guarded code, all catch blocks are ignored

Catching exceptions

E.g.

```
public static void main(String[ ] args) {
  int n = DEFAULT;
  try {
    n = Integer.parseInt(args[0]);
    if (n <= 0) {
      System.out.println("n must be positive. Using default.");
      n = DEFAULT;
    }
  } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("No argument specified for n. Using default.");
  } catch (NumberFormatException e) {
    System.out.println("Invalid integer argument. Using default.");
  }
}
```

Throwing exceptions

- Exceptions originate when a piece of Java code finds some sort of problem during execution and throws an exception object.
  - This is done by using the throw keyword followed by an instance of the exception type to be thrown.
  - It is often convenient to instantiate an exception object at the time the exception has to be thrown.

```
public void ensurePositive(int n) {
  if (n < 0)
    throw new IllegalArgumentException("That's not positive!");
  // ...
}
```

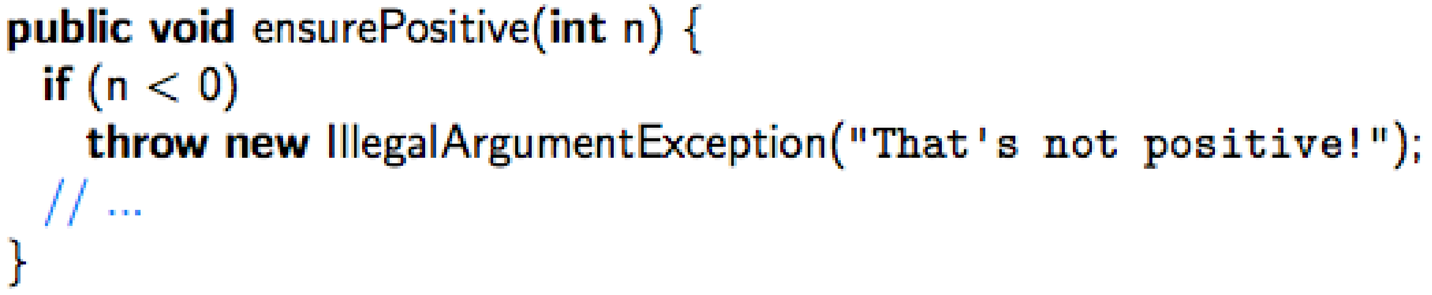
The throws clause

- When a method is declared, it is possible to explicitly declare, that an exception type may be thrown during a call to that method.
- The exception could be directly from a throw statement in that method body or propagated upward from a secondary method call made from within the body.
- For example, the parseInt method of the Integer class has the following formal signature:

```
public static int parseInt(String s) throws NumberFormatException;
```

Throwing exceptions

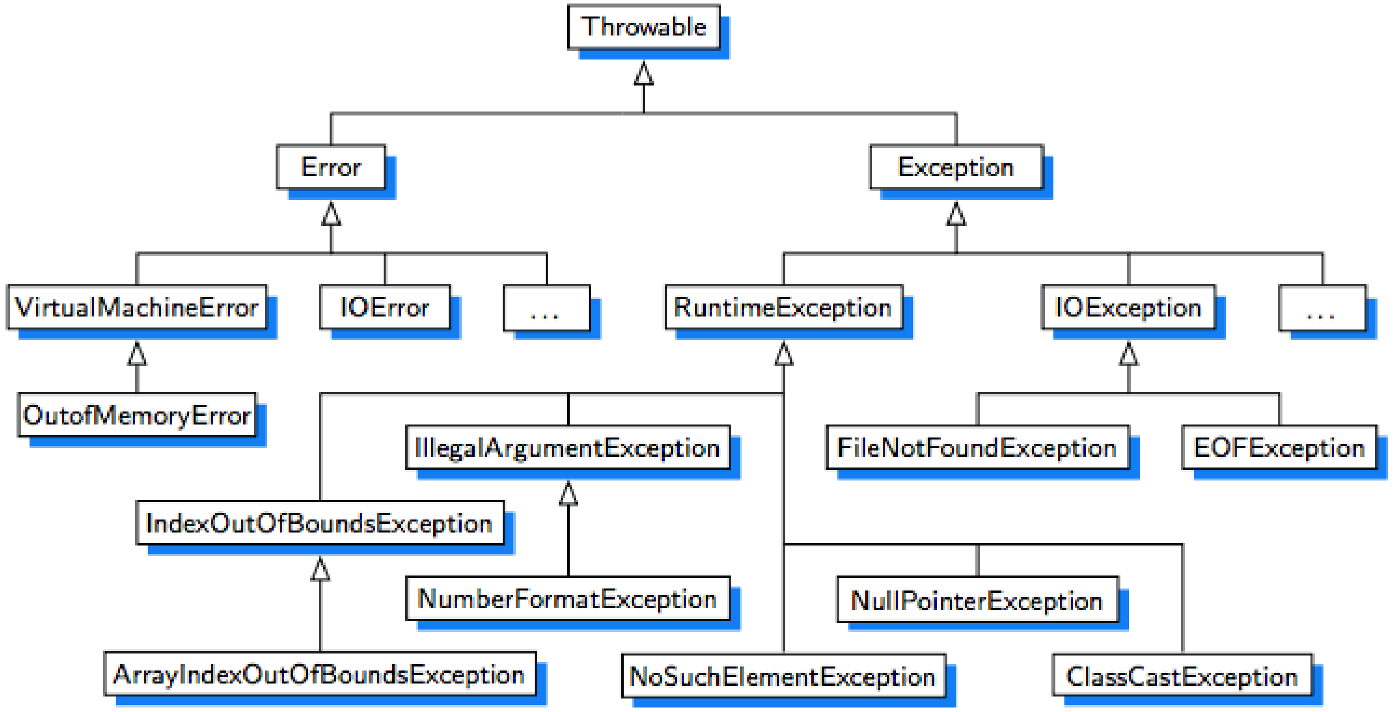
- **throw new** exceptionType(parameters);
  - where exceptionType is the type of the exception and the parameters are sent to that type’s constructor.
- E.g.



```
public void ensurePositive(int n) {
  if (n < 0)
    throw new IllegalArgumentException("That's not positive!");
  // ...
}
```

Java’s exception hierarchy

- Java defines a rich inheritance hierarchy of all objects that are deemed Throwable.
  - ***Errors*** are typically thrown only by the Java Virtual Machine and designate the most serious situations that are unlikely to be recoverable
    - e.g. when the virtual machine is asked to execute a corrupt class file, or when the system runs out of memory.
  - ***Exceptions*** designate situations in which a running program might reasonably be able to recover,
    - e.g. when unable to open a data file.


Throwable
Error
Exception
VirtualMachineError
IOError
. . .
RuntimeException
IOException
. . .
OutofMemoryError
IllegalArgumentException
FileNotFoundException
EOFException
IndexOutOfBoundsException
NumberFormatException
NullPointerException
ArrayIndexOutOfBoundsException
NoSuchElementException
ClassCastException

Checked and unchecked exceptions

- Java provides further refinement by declaring the RuntimeException class as an important subclass of Exception.
- All subtypes of RuntimeException in Java are officially treated as ***unchecked exceptions***, and any exception type that is not part of the RuntimeException is a ***checked exception.***

Checked and unchecked exceptions

- Runtime exceptions occur due to mistakes/bugs in programming logic.
- While such programming errors will certainly occur as part of the software development process, they should presumably be resolved before software reaches production quality.
- Therefore, it is not in the interest of efficiency to explicitly check for each such mistake at runtime, and thus these are designated as “unchecked” exceptions.

Checked and unchecked exceptions

- In contrast, other exceptions occur because of conditions that cannot easily be detected until a program is executing, such as an unavailable file or a failed network connection.
- Those are typically designated as “checked” exceptions in Java (and thus, not a subtype of RuntimeException).
- The designation between checked and unchecked exceptions plays a significant role in the syntax of the language.
  - In particular, ***all checked exceptions that might propagate upward from a method must be explicitly declared in its signature. Or, they must be caught. (Catch or Throws)***

Javadoc

In order to encourage good use of block comments and the automatic production of documentation, the Java programming environment comes with a documentation production program called *javadoc*.

Written in a format that allows the javadoc tool to parse the comments and extract information about the classes and methods

- part of the JDK tool set

Javadoc

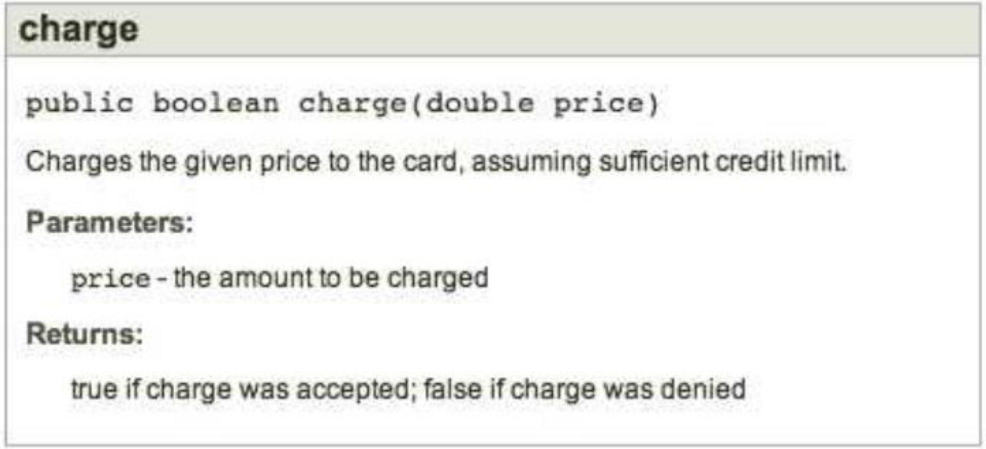
- Used to create online documentation in HTML about a set of classes.
  - Java API documentation is created using this technique
  - https://docs.oracle.com/javase/9/docs/api/
  - When changes are made to the API classes (and their comments) the javadoc tool is run again to generate the documentation
    - ensures that the documentation does not lag behind the evolution of the code

Javadoc

The primary javadoc tags that we use in this course:

- @author text: Identifies each author (one per line) for a class.
- @throws exceptionName description: Identifies an error condition that is signaled by this method
- @param parameterName description: Identifies a parameter accepted by this method.
- @return description: Describes the return type and its range of values for a method.

```
/**
 * Charges the given price to the card, assuming sufficient credit limit.
 * @param price  the amount to be charged
 * @return true  if charge was accepted; false if charge was denied
 */
public boolean charge(double price) {            // make a charge
  if (price + balance > limit)                   // if charge would surpass limit
    return false;                                // refuse the charge
  // at this point, the charge is successful
  balance += price;                              // update the balance
  return true;                                   // announce the good news
}
```



charge

`public boolean charge(double price)`

Charges the given price to the card, assuming sufficient credit limit.

Parameters:

`price` - the amount to be charged

Returns:

true if charge was accepted; false if charge was denied

Notes, figures and code adapted from

- **M. T. Goodrich and R. Tamassia:** *Data Structures and Algorithm in Java (6th Edition)*