Source = <https://www.cs.rit.edu/~hpb/Lectures/courses.html>

**Object-Oriented Ingredients**

| • |  | Objects |  |
| --- | --- | --- | --- |
| • |  | Class |  |
| • |  | Methods |  |
| • |  | Encapsulation |  |
| • |  | Inheritance |  |
| • |  | Polymorphism |  |
| • |  | Reuse |  |

**Object**

Objects are the things you think about first in designing a program and they are also the units of code that are eventually derived from the process.

In between, each object is made into a generic class of object and even more generic classes are defined so that objects can share models and reuse the class definitions in their code.

Each object is an instance of a particular class or subclass with the class’s own methods or procedures and data variables. An object is what actually runs in the computer.

**Class**

A class consists of all objects with like state which know exactly the same methods, i.e., a class knows the structure of its objects and contains the methods to manipulate the structure. An object is called an instance of a class.

**Encapsulation**

Encapsulation is the inclusion within a program object of all the resources need for the object to function — basically, the methods and the data.

Other objects adhere to use the object without having to be concerned with how the object accomplishes it.

The idea is "don’t tell me how you do it; just do it." An object can be thought of as a self-contained atom. The object interface consists of public methods and instance data.

**Methods**

A method is a programmed procedure that is defined as part of a class and included in any object of that class. A class (and thus an object) can have more then one method. A method in an object can only have access to the data known to that object, which ensures data integrity among the set of objects in an application. A method can be re-used in multiple objects.

| • |  | A class can have subclasses that can inherit all or some of the characteristics of the class. In relation to each subclass, the class becomes the superclass. |
| --- | --- | --- |
| • |  | Subclasses can also define their own methods and variables that are not part of their superclass. The structure of a class and its subclasses is called the class hierarchy. |

**Inheritance**

Inheritance is the concept that when a class of objects is defined, any subclass that is defined can inherit the definitions of one or more general classes.

This means for the programmer that an object in a subclass need not carry its own definition of data and methods that are generic to the class (or classes) of which it is a part.

This not only speeds up program development; it also ensures an inherent validity to the defined subclass object (what works and is consistent about the class will also work for the subclass)

**Polymorphism**

Polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning to a particular symbol or "operator" in different contexts.

**Reuse**

Do not reinvent the wheel, speed up the development, and reduces bugs. Existing classes have most likely no bugs.

**object-oriented**

Java is fully object oriented, single inheritance based.

**statically typed**

All data must be declared with types so that data and operations may be matched as far as possible during compilation. Methods are dynamically bound but overloaded signatures are decided during compilation. Like Objective C, Java permits type queries and object analysis, e.g., for the existence of methods, using the package java/lang/reflect at runtime.

**compiled**

A program is compiled into byte codes, the machine language of the Java Virtual Machine (JVM). The JVM is interpreted or compiled just in time. Classes and methods are bound symbolically, i.e., classes can be recompiled individually.

**architecture neutral**

Byte codes are the same everywhere; only the JVM has to be implemented for a new platform. Java programs should run everywhere and can be distributed as binaries. Unlike C and C++, Java completely specifies the capacity and behavior of the primitive data types thus eliminating a serious portability problem.

**multi-threaded**

Graphical user interfaces provide the illusion of parallel execution. Threads offer an elegant implementation. Java has a thread system based on classes and the language contains simple synchronization mechanisms (monitors). Many class libraries are thread-safe.

**garbage collected**

Dynamic memory management as in C and C++, where the programmer attends to reusing resources, is efficient but error prone. Java only knows dynamic objects and vectors and completely frees the programmer from the problem of memory reuse. Garbage collection runs as a parallel thread and thus should not be a bottleneck in critical situations.

**robust**

Exceptions are an integral part of Java for error handling. The programmer is constantly forced to consider error possibilities in libraries and the compiler can check that exceptions are not hidden or overlooked.

**secure**

An interpreter can pretty much ensure that the interpreted program cannot crash its platform. In connection with the Web, Java has additional security mechanisms that constrain foreign programs so that viruses are considered impossible — in spite of the fact that binary Java programs can run on arbitrary platforms.

**extensible**

Java methods can be implemented in other languages using the Java Native Interface (JNI). In principle, arbitrary libraries can be accessed as long as other security or portability aspects do not prevail.

**Identifier**

An [identifier](http://java.sun.com/docs/books/jls/html/3.doc.html)

is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter. An identifier cannot have the same spelling as a keyword, Boolean literal, or the null literal.

<https://www.tutorialspoint.com/difference-between-identifier-and-variable>

Key Differences between Identifier and Variable

1. Both an identifier and a variable are the names allotted by users to a particular entity in a program. The identifier is only used to identify an entity uniquely in a program at the time of execution whereas, a variable is a name given to a memory location, that is used to hold a value.
2. Variable is only a kind of identifier, other kinds of identifiers are function names, class names, structure names, etc. So it can be said that all variables are identifiers whereas, vice versa is not true.

**why we can't call .length on null string java**

Null isn't really a value - and as such, doesn't have properties. So, calling anything that is meant to return a value - such as . length() , will invariably return an error, because 'nothing' cannot have properties.

• Strings are constants

• All String literals are instances of the String class and exist once in a JVM.

1 class StringLiteral {

2

3 public static void main( String args[] ) {

4 String aString = "you";

5 String bString = "yo" + "u"; // compiler

6 String cString = "you";

7

8 if ( cString == aString )

9 System.out.println("1. equal");

10 if ( cString.equals(aString) )

11 System.out.println("2. equal");

12 if ( "you".equals(aString) )

13 System.out.println("3. equal");

14 if ( bString.equals(aString) )

15 System.out.println("4. equal");

16 if ( "yo" + "u" == aString )

17 System.out.println("5. ==");

18 if ( bString == aString )

19 System.out.println("6. ==");

20 if ( bString == new String("you") )

21 System.out.println("7. ==");

22 else

23 System.out.println("8. !=");

24

25 }

26 }

Source Code: [Src/4/StringLiteral.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/4/StringLiteral.java)

Result:

1. equal

2. equal

3. equal

4. equal

5. ==

6. ==

8. !=

The values of of variable of type String are immutable.

[Why?](https://www.programcreek.com/2013/04/why-string-is-immutable-in-java/)

Other Example

1 /\*\*

2 \* "abc" versus new String("abc")`

3 \*/

4

5 class StringL {

6

7 public static void method(String id, String literal, String aNewString) {

8 System.out.println(id + " in method");

9 System.out.print("\tliteral= aNewString\n ");

10 System.out.println( literal == aNewString);

11 }

12 public static void main( String args[] ) {

13 String aString = "abc";

14 System.out.print("abc == aString\n ");

15 System.out.println("abc" == aString);

16

17 String newString = new String("abc");

18 System.out.print("abc == new String(abc)\n ");

19 System.out.println("abc" == newString);

20

21 method("1", "abc", "abc");

22 method("2", "abc", new String("abc") );

23 method("3", "abc", "ab" + "c");

24 method("4", "abc", "" + "abc");

25 }

26 }

Source Code: [Src/4/StringL.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/4/StringL.java)

Output:

abc == aString

true

abc == new String(abc)

false

1 in method

literal= aNewString

true

2 in method

literal= aNewString

false

3 in method

literal= aNewString

true

4 in method

literal= aNewString

True

**From a Previous Bridge Exam = Now available = theSolution Up to You**

1 public class X\_s1 {

2

3 public static void method\_c() {

4 String right = new String(10 \* 1 - 10 + "");

5 String middle = "00".substring(0, 1);

6 String zero = "0";

7 String left = "1";

8 String leftLeft = "2";

9 int aInt = 3;

10

11 System.out.println("c\_0: " + ( "0" == zero ) );

12 System.out.println("c\_1: " + ( "0" == left ) );

13 System.out.println("c\_2: " + ( "0" == middle ) );

14 System.out.println("c\_3: " + ( aInt + aInt ) );

15 System.out.println("c\_4: " + ( left + aInt + aInt ));

16 System.out.println("c\_5: " + ( ( left = left + aInt + aInt ) ) + left );

17 // c\_3: 6

18 // c\_4: 133

19 // c\_5: 133133

20 System.out.println("c\_6: " + ( ( leftLeft = left = ( left = left + aInt + aInt ) ) ) + left );

21 System.out.println("c\_7: " + ( leftLeft ) );

22 System.out.println("c\_7: " + ( ( "1" + "" ) == "1" ) );

23 // c\_6: 1333313333

24 // c\_7: 13333

25 // c\_7: true

26 System.out.println("c\_8: " + ( ( aInt + "1" ) == ( left + "1" ) ) );

27 // c\_8: false

28 System.out.println( 1 + "1" );

29 System.out.println( ( 1 + 1 ) + "1" );

30 // 11

31 // 21

32 }

33 public static void main(String argv[]) {

34 method\_c();

35 }

36 }

OUTPUT:-

c\_0: true

c\_1: false

c\_2: false

c\_3: 6

c\_4: 133

c\_5: 133133

c\_6: 1333313333

c\_7: 13333

c\_7: true

c\_8: false

11

21

class test {

public static void main(String args[]) {

String aString = "0";

String bString = "0" + "1";

System.out.println(aString + "1" == "01" );

System.out.println((aString + "1").equals("01") );

}

}

OUTPUT:-

false

true

Why is it not not true in all cases?

**== vs equal in java**

The == operator tests whether two variables have the same references (aka pointer to a memory address). Whereas the equals() method tests whether two variables refer to objects that have the same state (values).

Other example:

1 class Literal\_1 {

2

3 public static void main( String args[] ) {

4 int three = 3;

5 String aString = "12";

6

7 if ( ("hel" + "lo") == "hello" )

8 System.out.println("1");

9 if ( ("hel" + "lo").equals("hello") )

10 System.out.println("2");

11 if ( ( aString + three).equals("123") )

12 System.out.println("3");

13 if ( ( aString + three) == "123" )

14 System.out.println("4");

15 }

16 }

**More on Strings**

1 /\*\*

2 \* Play with the String class

3 \*

4 \* @version $Id$

5 \*

6 \* @author Hpb

7 \*

8 \* $Log$

9 \*/

10

11 class StringUse {

12

13 public static void compare(String aString, String bString) {

14 if ( aString.equals(bString) )

15 System.out.println("\tequal");

16 else

17 System.out.println("\t! equal");

18 if ( aString == bString)

19 System.out.println("\t== ");

20 else

21 System.out.println("\t! ==");

22

23 }

24 public static void main( String args[] ) {

25 String aString = "David";

26 String bString = "David";

27 compare(aString, bString);

28

29 System.out.println("Using New");

30 aString = new String("David");

31 bString = new String("David");

32 compare(aString, bString);

33

34 System.out.println("Concatenation 1");

35 aString = "Da" + "vid";

36 bString = "" + "David";

37 compare(aString, bString);

38

39 System.out.println("Concatenation 2");

40 aString = "Da" + "vid";

41 bString = "D" + "a" + "vid";

42 compare(aString, bString);

43

44 }

45 }

Source Code: [Src/4/StringUse.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/4/StringUse.java)

Execution:

% java StringUse

equal

==

Using New

equal

! ==

Concatenation 1

equal

==

Concatenation 2

equal

==

**QUESTION - WHAT HAPPENS IF YOU DELETE THE “default” line in switch statement?**

/\*\*

\* Test of the switch statement.

\*

\* @version $Id$

\*

\* @author hpb

\*

\* Revisions:

\* $Log$

\*/

class test {

private String itWasA(char c) {

switch( c ) {

case 'a': return("A"); // break?

case 'b': return("B"); // break?

case 'c': return("C"); // break?

case 100: return("D"); // break?

case 101: return("E"); // break?

default: return("no clue, but not an [a-e]");

// What happens if

// we delete this line?

}

}

public static void main( String args[] ) {

char theChar;

theChar = 'd';

System.out.println("You typed in an '" +

new test().itWasA(theChar) + "'");

System.exit(0); // beware of ...

}

}

**ANSWER** - gives an error:

java: missing return statement

**Pattern**

See <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/regex/Pattern.html>

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/regex/Matcher.html>

**Example**

1 import static java.lang.Math.\*;

2 public class CmToInch {

3 static final double centimeterToInchMulitplier = 2.54;

4 static CmToInch aCmToInch = null; // use before define

5 // assign object before use

6 double inCentimeter = 0;

7 double inInch; // = 0;

8 static double totalCentimeter;

9 static double totalInch;

10

11

12 public CmToInch() {

13 }

14 public CmToInch(int inInch) {

15 this.inInch = inInch;

16 }

17 public void cm(double soLong){

18 inCentimeter = soLong;

19 inInch = inCentimeter \* centimeterToInchMulitplier;

20 totalCentimeter += inCentimeter;

21 totalInch += inInch;

22

23 }

24 public String toString(){

25 return "centimeter/inch = " + inCentimeter + "/" + inInch;

26 }

27 public static void total(){

28 System.out.println(" totalCentimeter " + totalCentimeter);

29 System.out.println(" totalInch " + totalInch);

30 }

31

32 public static void main(String args[] ) {

33 System.out.println("pi = " + PI); // from where

34 System.out.println("e = " + Math.E);

35 CmToInch aCmToInch = new CmToInch(); // which one

36 CmToInch aaCmToInch = new CmToInch();

37

38 aCmToInch.cm(1);

39 aaCmToInch.cm(10);

40 System.out.println(aaCmToInch);

41 total();

42 new CmToInch().total();

43 }

44 }

Source Code: [Src/5/CmToInch.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/5/CmToInch.java)

Execution:

% java CmToInch

pi = 3.141592653589793

e = 2.718281828459045

centimeter/inch = 10.0/25.4

totalCentimeter 11.0

totalInch 27.939999999999998

totalCentimeter 11.0

totalInch 27.939999999999998

**Static in Classes/Methods Lifetime**

• Class Variables: If a variable is declared static, there exists exactly one incarnation of the field,

• Static Methods: A method that is declared static is called a class method. A class method is always invoked without reference to a particular object.

• Non Static Methods: A method that is not declared static is called an instance method, and sometimes called a non-static method. An instance method is always invoked with respect to an object, which becomes the current object to which the keywords this and super refer during execution of the method body.

• Variables can be declared:

static: class variable

— final: can be assigned once or zero time

— transient: not stored or saved via the standard serialization process

— volatile: A variable may be modified by multiple threads. This gives a hint to the compiler to fetch the value each time, rather store a locale copy. This also prohibits same optimization procedures.

• See also: <http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#78119>

1

2 public class Overview {

3 int instanceVariable;

4 static int classVariable;

5 final int finalVariable; // static?

6 volatile int volatileVariable;

7 transient int transientVariable;

8

9

10 public Overview() {

11 finalVariable = 42;

12 }

13 public Overview(int aLocalVariable) {

14 finalVariable = 43;

15 }

16 void instanceMethod() {

17 finalVariable = 43;

18 instanceVariable = 22;

19 classVariable = 33;

20 }

21 static void classMethod() {

22 classVariable = 3;

23 }

24

25 public static void main(String args[] ) {

26 Overview aOverview = new Overview();

27 Overview bOverview = new Overview();

28 Overview cOverview = new Overview(1);

29 cOverview = bOverview;

30 aOverview.instanceMethod();

31 instanceMethod();

32 bOverview.classMethod();

33 // values of aOverview.? bOverview.?

34 // aOverview.finalVariable??

35

36 }

37 }

Source Code: [Src/5/Overview.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/5/Overview.java)

• how many instances of the variables to exist?

• How many objects do exist?

**Additional Examples**

See <http://docs.oracle.com/javase/specs/jls/se7/html/index.html>

1 public class Scope\_1 {

2

3 String aString = null;

4

5 public void method(String aString) {

6 this.aString = aString;

7 }

8 public Scope\_1 test() {

9 String aString = new String("set in test");

10

11 method(aString);

12 return this;

13 }

14 public static void main(String args[] ) {

15 System.out.println((new Scope\_1().test()).aString);

16

17 }

18 }

Source Code: [Src/5/Scope\_1.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/5/Scope_1.java)

EXECUTION:-

set in test

**Inheritance**

See also [here.](http://java.sun.com/docs/books/jls/html/8.doc.html#30229)

Java has simple inheritance, i.e., a class can only extend one superclass.

Class Members

The members of a class type are all of the following:

|  | • |  | Members inherited from its direct superclass, except in class Object, which has no direct superclass |
| --- | --- | --- | --- |
|  | • |  | Members inherited from any direct super interfaces |
|  | • |  | Members declared in the body of the class. |
|  | • |  | Is a relationship |

Members of a class that are declared private are not inherited by subclasses of that class. Only members of a class that are declared protected or public are inherited by subclasses declared in a package other than the one in which the class is declared.

Constructors and static initializers are not members and therefore are not inherited.

**Constructor Sequence**

1 public class B {

2

3 int aBint;

4

5 public B() {

6 System.out.println("public B()" );

7 }

8 public B(int aBint) {

9 this.aBint = aBint;

10 System.out.println("public B(int aBint)" );

11 }

12 public String toString() {

13 return "" + getClass();

14 }

15 public static void main(String args[]) {

16 System.out.println("1: " + new B());

17 System.out.println("2: " + new B(42));

18 }

19 }

20

Source Code: [Src/6/B.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/B.java)

Next

1 class BB extends B {

2

3 int aBBint;

4 public int x;

5

6 public BB() {

7 System.out.println("public BB()" );

8 }

9 public BB(int aBBint) {

10 this.aBBint = aBBint;

11 System.out.println("public BB(int x)" );

12 }

13 public String toString() {

14 return "" + getClass();

15 }

16 public static void main(String args[]) {

17 System.out.println("1: " + new BB());

18 System.out.println("2: " + new BB(42));

19 }

20 }

Source Code: [Src/6/BB.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/BB.java)

Next

% java BB

public B()

public BB()

1: class BB

public B()

public BB(int x)

2: class BB

The Flock class:

1 public class Flock {

2

3 private static int soManyBirds = 0;

4 static final int maxNumberOfBirds = 3;

5

6 public Flock() {

7 soManyBirds ++;

8 }

9 public int soManyBirds() {

10 return soManyBirds;

11 }

12 public String toString() {

13 return "" + soManyBirds;

14 }

15 }

Source Code: [Src/6/Flock.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/Flock.java)

The pigeon class

1 public class Pigeon extends Flock {

2

3 private String name;

4

5 public Pigeon() {

6 }

7 public Pigeon(String name) {

8 this.name = name;

9 }

10 private static void test() {

11 int soManyBirdsCreated = 0;

12 String name = "a";

13 Pigeon lastOne = null; // why?

14 while ( soManyBirdsCreated++ < maxNumberOfBirds ) {

15 lastOne = new Pigeon(name += "a");

16 }

17 System.out.println("so many are in the flock: " + lastOne.soManyBirds());

18 System.out.println("last name used = " + name );

19 }

20 public static void main(String[] args ) {

21 test();

22 }

23 }

Source Code: [Src/6/Pigeon.jav](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/Pigeon.java)a

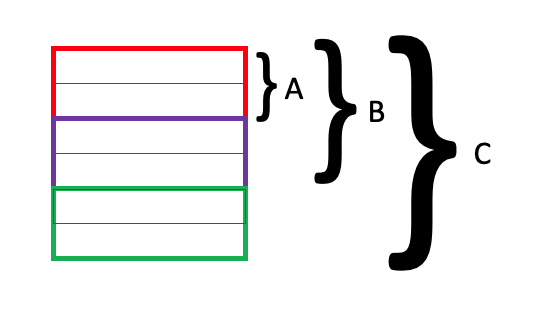
EXECUTION:-

so many are in the flock: 3

last name used = aaaa

class C extends class B extends class A

A picture:



**How to get access to super classmethods/variables?**

• Super class:

1 public class S {

2

3 public int intS; // what is the value of intS?

4

5 public S () {

6 System.out.println("S()");

7 }

8 public S method(int x) {

9 intS = x;

10 System.out.println("S method(int x)");

11 return this;

12 }

13 public String toString() {

14 return "S: " + intS;

15 }

16 public static void main(String args[]) {

17 System.out.println("new S() " + new S());

18 }

19 }

Source Code: [Src/6/S.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/S.java)

• Sub class:

1 public class SubclassOfS extends S {

2

3 public int intS;

4

5 public SubclassOfS() {

6 System.out.println("SubclassOfS ()");

7 }

8

9 public S method(int x) {

10 intS = x;

11 System.out.println("S method(int x)");

12 super.method(9);

13 System.out.println("4. super: " + super.toString() );

14 super.intS = 4;

15 System.out.println("5. super: " + super.toString() );

16 return this;

17 }

18 public String toString() {

19 return "SSubclassOfS: " + intS;

20 }

21

22

23 public static void main(String args[]) {

24 SubclassOfS aSubclassOfS = new SubclassOfS();

25 S aS = aSubclassOfS.method(42);

26 System.out.println(aS);

27 System.out.println(aSubclassOfS);

28 System.out.println("1. SubclassOfS!intS = " + aSubclassOfS.intS);

29 System.out.println("2. ((S)SubclassOfS)!intS = " + ((S)aSubclassOfS).intS);

30 // method(3); // <--- what is the problem here ...

31

32 }

33 }

Source Code: [Src/6/SubclassOfS.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/SubclassOfS.java)

Execution:

% java SubclassOfS

S()

SubclassOfS ()

S method(int x)

S method(int x)

4. super: S: 9

5. super: S: 4

SSubclassOfS: 42

SSubclassOfS: 42

1. SubclassOfS!intS = 42

2. ((S)SubclassOfS)!intS = 4

**Default Constructor**

• Constructors create objects from the class blueprint

• The default constructor has no arguments

• Constructors can invoke, as the first statement, other constructors

• Constructor declarations look like method declarations, except that they use the name of the class and have no return type.

• The compiler will provide a default constructor if none is defined. The compiler will then verify that the super class does have provided default constructor

• The compiler will give an error, if a parameterized constructor is defined, but no default constructor is defined, and you use the provided constructor. This makes sense because you never wanted to allow to create an ’empty’ object.

**Determining Accessibility**

A member type or a constructor of a class type is accessible only if the type is accessible and the member or constructor is declared to permit access:

|  | • |  | If the member or constructor is declared public, then access is permitted. |
| --- | --- | --- | --- |
|  | • |  | Protected methods and variables are accessible to subclasses and provide a way of opening up the encapsulation. |
|  | • |  | Private methods and variables are not accessible to subclasses. |
|  | • |  | A final class can not be sub classed. |

**Polymorphism**

A subclass can override an inherited method by providing a new, identical method declaration.

**VERY IMPORTANT EXAMPLE:-**

public class Vehicle {

String honkSound = "vehicle honk";

int wheels = 4;

public void setSound (String honkSound) {

this.honkSound = honkSound;

}

public String toString () {

return "v";

}

public void setSoManyWheels (int wheels) {

this.wheels = wheels;

}

public int soManyWheels () {

return wheels;

}

public void honk() {

System.out.println(honkSound);

}

}

public class Train extends Vehicle {

String honkSound = "choo-choo";

int wheels = 32;

public void setSound (String honkSound) {

this.honkSound = honkSound;

}

public int soManyWheels () {

return wheels;

}

public void onlyAtrainCanDoThis() {

}

public void honk() {

System.out.println(honkSound);

}

public void de() {

System.out.println("de");

}

public static void main(String[] args ) {

new Vehicle().honk();

new Train().honk();

new Train().de();

}

}

public class TrainEx {

String honkSound = "choo-choo";

public void honk() {

System.out.println(honkSound);

}

public static void main(String[] args ) {

new Vehicle().honk();

new Train().honk();

Train aTrain = new Train();

Vehicle aVehicle = aTrain; // new Vehicle();

Vehicle aVehicleOtherReference = aVehicle;

System.out.println(aVehicle + " " +aTrain);

System.out.println(aVehicle + " " +aVehicle);

aVehicle.honk();

aTrain.honk();

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

aVehicle.setSound("ringADing");

aVehicle.honk();

aTrain.honk();

System.out.println("aVehicle.honkSound: " + aVehicle.honkSound);

System.out.println("aTrain.honkSound: " + aTrain.honkSound);

}

}

**OUTPUT**:- (what i had thought)

Vehicle honk

Choo choo

V v

V v

Vehicle honk

Choo choo

—-------------------------

Ringading

Choo choo

Ringading

Choo choo

**ACTUAL OUTPUT**:-

vehicle honk

choo-choo

v v

v v

choo-choo

choo-choo

--------------------

ringADing

ringADing

aVehicle.honkSound: vehicle honk

aTrain.honkSound: ringADing

**VERY IMPORTANT LESSON**:-

“aVehicle” (- an instance of child class typecasted to parent class) object can access methods of the “Train” class but not its parameters/variables. It uses its own variables.

Child class looks for parameters in its own class, if not present then looks up in parent class. But an instance of child class typecasted to parent class, can access only the methods of the child, but the parameters of parent only.

**7.12. Class Cast - II**

Vehicle:

1 public class Vehicle {

2

3 String honkSound = "vehicle honk";

4 int wheels = 4;

5

6 public void setSound (String honkSound) {

7 this.honkSound = honkSound;

8 }

9 public String toString () {

10 return "v";

11 }

12 public void setSoManyWheels (int wheels) {

13 this.wheels = wheels;

14 }

15 public int soManyWheels () {

16 return wheels;

17 }

18 public void honk() {

19 System.out.println(honkSound);

20 }

21 }

22

Source Code: [Src/6/Vehicle.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/Vehicle.java)

Train:

1 public class Train extends Vehicle {

2

3 String honkSound = "choo-choo";

4 int wheels = 32;

5

6 public void setSound (String honkSound) {

7 this.honkSound = honkSound;

8 }

9 public int soManyWheels () {

10 return wheels;

11 }

12 public void onlyAtrainCanDoThis() {

13 }

14 public void honk() {

15 System.out.println(honkSound);

16 }

17 public void de() {

18 System.out.println("de");

19 }

20 public static void main(String[] args ) {

21 new Vehicle().honk();

22 new Train().honk();

23 new Train().de();

24 }

25 }

26

Source Code: [Src/6/Train.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/Train.java)

Casting:

1 public class Casting {

2

3 String honkSound = "choo-choo";

4

5 public void honk() {

6 System.out.println(honkSound);

7 }

8 public static void main(String[] args ) {

9 new Vehicle().honk();

10 new Train().honk();

11

12 Train aTrain = new Train();

13 Vehicle aVehicle = (Train)aTrain;

14 Vehicle aVehicleOtherReference = aVehicle;

15 System.out.println(aTrain + " " +aTrain);

16 System.out.println(aVehicle + " " +aVehicle);

17 aVehicle.honk();

18 System.out.println("aVehicle.soManyWheels() = " + aVehicle.soManyWheels());

19 aVehicle.honk();

20 aVehicle.setSound("ring ring");

21 aTrain.setSound("ring ring");

22 aVehicle.honk();

23 aTrain.honk();

24

25 // aVehicle.onlyAtrainCanDoThis();

26 aVehicle.wheels = 0;

27 System.out.println("aVehicle.soManyWheels() = " + aVehicle.soManyWheels());

28 System.out.println("aTrain.soManyWheels() = " + aTrain.soManyWheels());

29 System.out.println("aVehicleOtherReference.soManyWheels() = " + aVehicleOtherReference.soManyWheels());

30 System.out.println("aVehicle.wheels = " + aVehicle.wheels);

31 }

32 }

Source Code: [Src/6/Casting.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/Casting.java)

OUTPUT:-

vehicle honk

choo-choo

v v

v v

choo-choo

aVehicle.soManyWheels() = 32

choo-choo

ring ring

ring ring

aVehicle.soManyWheels() = 32

aTrain.soManyWheels() = 32

aVehicleOtherReference.soManyWheels() = 32

aVehicle.wheels = 0

**Abstract Classes**

An abstract class

• specifies a public method interface which can be inherited by direct or indirect subclasses.

• may declare methods, but not implement them.

• can not be instantiated.

Classes who extend an abstract class share the same, possibly extended, interface.

• Is a relationship

Next

1 /\*\*

2 \* Abstract class

3 \* @version $Id$

4 \*

5 \* @author hp bischof

6 \*

7 \* Revisions:

8 \* $Log$

9 \*/

10

11 abstract class Area extends Object {

12

13 String type;

14

15 public String getType() {

16 return type;

17 }

18

19 public abstract int area();

20 public abstract int perimeter();

21 }

Source Code: [Src/6b/Area.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6b/Area.java)

Next

1 /\*\*

2 \* This class implements a Circle class.

3 \*

4 \* @version $Id$

5 \*

6 \* @author hp bischof

7 \*

8 \* Revisions:

9 \* $Log$

10 \*/

11

12 public class Circle extends Area {

13 private int radius;

14 public Circle(int \_radius) {

15 type = "Circle";

16 radius = \_radius;

17 }

18

19 public int area() {

20 return (int)(Math.PI \* radius \* radius);

21 }

22 // /\*

23 public int perimeter() {

24 return (int)(Math.PI \* radius \* radius);

25 }

26 // \*/

27 }

Source Code: [Src/6b/Circle.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6b/Circle.java)

You will get a compiler error, if a class doesn’t implement all methods.

% javac C\*a

Circle.java:12: class Circle must be declared abstract.

It does not define int perimeter() from class Area.

public class Circle extends Area {

^

1 error

1 /\*\*

2 \* This class implements a Square class.

3 \*

4 \* @version $Id$

5 \*

6 \* @author hp bischof

7 \*

8 \* Revisions:

9 \* $Log$

10 \*/

11

12 public class Square extends Area {

13

14 private int length;

15

16 public Square(int \_length) {

17 type = "Square";

18 length = \_length;

19 }

20

21 public int area() {

22 return length \* length;

23 }

24

25 public int perimeter() {

26 return 4 \* length;

27 }

28

29 }

Source Code: [Src/6b/Square.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6b/Square.java)

**Class Cast and Abstract Classes**

1 abstract class A {

2

3 public int x;

4

5 abstract public A a(int x);

6

7 public A aa(int x) {

8 System.out.print("- in A!aa");

9 return this;

10 }

11

12 }

Source Code: [Src/6/A.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/A.java)

Next

1 class AX extends A {

2

3 public int x;

4

5 public A a(int x) {

6 System.out.print("= in AX!a");

7 return this;

8 }

9

10 public static void main(String args[]) {

11 AX aAX = new AX();

12 A aA = (A)aAX;

13

14 System.out.println("aAX.a(42) " + aAX.a(42) );

15 System.out.println("aAX.a(43) " + aAX.aa(43) );

16

17 System.out.println("aA.aa(44) " + aA.aa(44) );

18 System.out.println("aA.a(45) " + aA.a(45) ); // <--

19 }

20 }

Source Code: [Src/6/AX.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/AX.java)

Next

% java AX

= in AX!aaAX.a(42) AX@e76cbf7

- in A!aaaAX.a(43) AX@e76cbf7

- in A!aaaA.aa(44) AX@e76cbf7

= in AX!aaA.a(45) AX@e76cbf7

**Interfaces**

• An interface specifies which methods must be implement.

• An interface defines an public API.

• This means, we can make sure, that unrelated classes share the same part of the interface.

• An interface defines constants. They are public, static and final regardless of wether these modifiers have been specified.

• Methods implementations have been added to be able to extend the functionality by modifying the interface and not the implementations.

• Interface methods can’t be native, static, synchronized, final, private, or protected

• Abstract and native methods can’t have a body.

• Fields in a field a static and final.

**Differences between Abstract Classes andInterfaces**

From: https://www.tutorialspoint.com/differences-between-abstract-class-and-interface-in-java

|  | Supported Methods |  |
| --- | --- | --- |

- Abstract class: can have both an abstract as well as concrete methods.

- Interface: can have only abstract methods. Java 8 onwards, it can have default as well as static methods.

|  | Multiple Inheritance |  |
| --- | --- | --- |

- Abstract class: Multiple Inheritance is not supported.

- Interface: supports multiple inheritance.

|  | Supported Variables |  |
| --- | --- | --- |

- Abstract class: final, non-final, static and non-static variables supported.

- Interface: Only static and final variables are permitted.

|  | Implementation |  |
| --- | --- | --- |

- Abstract class can implement an interface.

- Interface: can not implement an interface, it can extend an interface.

|  | Inheritance |  |
| --- | --- | --- |

- Abstract class: can inherit another class using extends keyword and implement an interface.

- Interface: can inherit only an interface.

|  | Inheritance |  |
| --- | --- | --- |

- Abstract class: can be inherited using extends keyword.

- Interface: can only be implemented using implements keyword.

|  | Access to Members |  |
| --- | --- | --- |

- Abstract class: can have any type of members like private, public.

- Interface: can only have public members.

**Aggregation**

• Aggregation is a design term, which means that you create an new object by composing it out of the others.

• Aggregation relationships are specified by classes and reflected by their instance objects.

• For example: A Cylinder class can be defined as:

1 /\*\*

2 \* This class implements a Cylinder Class

3 \* NOT COMPLETE

4 \* @version $Id$

5 \*

6 \* @author hp bischof

7 \*

8 \* Revisions:

9 \* $Log$

10 \*/

11

12 public class Cylinder {

13 private aCircle;

14 private aRect;

15

16 public Cylinder(int \_radius, \_height) {

17 aCircle = new Circle(radius);

18 aRect = new Rectangle(aCircle.perimeter(), height);

19 }

20 public int area() {

21 return aCircle.area \* 2 + aRect.area();

22 }

23 ....

24 }

Source Code: [Src/6b/Cylinder.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6b/Cylinder.java)

**Default Constructor Sequence**

Which constructor is called when?

1 public class X\_1 {

2

3 public X\_1() {

4 System.out.println(" in X\_1!X\_1()");

5 }

6

7 public X\_1(int x) {

8 System.out.println(" in X\_1!X\_1(int x)");

9 }

10

11 public X\_1(int x, int y) {

12 System.out.println(" in X\_1!X\_1(int x, int y)");

13 }

14

15 }

Source Code: [Src/6g/X\_1.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6g/X_1.java)

1 class X\_2 extends X\_1 {

2

3 public X\_2() {

4 // super(); // default

5 System.out.println(" in X\_2!X\_2()");

6 }

7 public X\_2(int x) {

8 // super(); // default

9 super(x);

10 System.out.println(" in X\_2!X\_2(int x)");

11 }

12

13 public X\_2(int x, int y) {

14 // super(); // default

15 System.out.println(" in X\_2!X\_2(int x, int y)");

16 }

17

18 public static void main(String args[])

19 {

20 X\_2 aX\_2 = new X\_2();

21 X\_2 aaX\_2 = new X\_2(3);

22 X\_2 aaaX\_2 = new X\_2(3, 3);

23 }

24 }

25

Source Code: [Src/6g/X\_2.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6g/X_2.java)

Result:

in X\_1!X\_1()

in X\_2!X\_2()

in X\_1!X\_1(int x)

in X\_2!X\_2(int x)

in X\_1!X\_1()

in X\_2!X\_2(int x, int y)

**Constructor must match**

Superclass has no *default()* constructor!

1 public class X\_1 extends Object {

2

3 public X\_1(int x) {

4 System.out.println(" in X\_1!X\_1(int x)");

5 }

6

7 }

Source Code: [Src/6h/X\_1.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6h/X_1.java)

1 class X\_2 extends X\_1 {

2

3 public X\_2() {

4 System.out.println(" in X\_2!X\_2()");

5 }

6

7 public static void main(String args[])

8 {

9 X\_2 aX\_2 = new X\_2();

10 }

11 }

12

Source Code: [Src/6h/X\_2.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6h/X_2.java)

Result:

% javac X\*a

X\_2.java:3: No constructor matching X\_1() found in class X\_1.

public X\_2() {

^

1 error

• Overloading of constructors is identical in behavior to overloading of methods. The overloading is resolved at compile time by each class instance creation expression.

• If a class contains no constructor declarations, then a default constructor that takes no parameters is automatically provided:

• If the class being declared is the primordial class Object, then the default constructor has an empty body.

• Otherwise, the default constructor takes no parameters and simply invokes the superclass constructor with no arguments.

• A compile-time error occurs if a default constructor is provided by the compiler but the superclass does not have a constructor that takes no arguments.

**Methods**

Access of methods and *super* methods.

1 public class X\_1 {

2

3 public X\_1() {

4 System.out.println(" in X\_1!X\_1()");

5 }

6

7 public X\_1(int x) {

8 System.out.println(" in X\_1!X\_1(int x)");

9 }

10

11 public void a() {

12 System.out.println(" in X\_1!a()");

13 }

14

15 }

Source Code: [Src/6f/X\_1.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6f/X_1.java)

1 class X\_2 extends X\_1 {

2

3 public X\_2() {

4 // super(); // default

5 super.a();

6 System.out.println(" in X\_2!X\_2()");

7 }

8 public X\_2(int x) {

9 // super(); // default

10 super(x);

11 System.out.println(" in X\_2!X\_2(int x)");

12 }

13

14 public void a() {

15 super.a();

16 System.out.println(" in X\_2!a()");

17 }

18

19

20 public static void main(String args[])

21 {

22 X\_2 aX\_2 = new X\_2();

23 aX\_2.a();

24 X\_2 anOtherX\_2 = new X\_2(3);

25 }

26 }

27

Source Code: [Src/6f/X\_2.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6f/X_2.java)

Result:

% java X\_2

in X\_1!X\_1()

in X\_1!a()

in X\_2!X\_2()

in X\_1!a()

in X\_2!a()

**Nested Classes**

• Nested classes can be: non-static or static nested.

• Non-static nested classes are referred to as inner classes.

• A nested class is a class that is a member of another class.

class Outer{

. . .

class AnestedClass {

. . .

}

}

• A nested class can be declared static or not.

• A static nested class is called a static nested class.

• A non static nested class is called an inner class.

• As with static methods and variables, a static nested class is associated with its enclosing class.

• And like class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class-it can use them only through an object reference.

• As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object’s instance variables and methods.

• Because an inner class is associated with an instance, it cannot define any static members itself.

• Example:

1 public class NestedClassEx {

2

3 public int inNestedClass;

4

5 void inNestedClass() {

6 System.out.println("NestedClass!inNestedClass");

7 (new AinnerClass()).aInnerClassM2();

8 }

9

10 static class AstaticClass {

11 static void aStaticClassM1() {

12 System.out.println("AstaticClass!aStaticClassM1");

13 }

14 void aStaticClassM2() {

15 System.out.println("AstaticClass!aStaticClassM2");

16 }

17 }

18

19 class AinnerClass {

20 /\*

21 static void aInnerClassM1() {

22 System.out.println("AinnerClass!aInnerClassM1");

23 }

24 NestedClassEx.java:15: inner classes cannot have static declarations

25 static void aInnerClassM1() {

26 \*/

27

28 void aInnerClassM2() {

29 System.out.println("AinnerClass!aInnerClassM2");

30 }

31 }

32

33 public static void main(String args[]) {

34

35 AstaticClass.aStaticClassM1();

36 (new AstaticClass()).aStaticClassM2();

37

38 (new NestedClassEx()).inNestedClass();

39 // (new AinnerClass()).aInnerClassM2();

40 }

41 }

Source Code: [Src/6/NestedClassEx.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6/NestedClassEx.java)

OUTPUT:-

AstaticClass!aStaticClassM1

AstaticClass!aStaticClassM2

NestedClass!inNestedClass

AinnerClass!aInnerClassM2

IMPORTANT LESSON:

If you are accessing a static method in a static class, then you have to call accordingly (which is without an instance creation of the class - AstaticClass.**aStaticClassM1**() )

But if you are accessing a non-static method in a static class, then you have to create an instance of class ( (new AstaticClass()).**aStaticClassM2**() ).

If you trying to access a non static class’s element/method, then you have to use a function which calls the nested inner class. (line 38 and 40)

**Static Initializer Blocks**

• Static initializer blocks are primarily used for initialization.

• The code in a static initializer block is executed when the class is initialized/

• A class can have more than one static initializer block.

• S.java:

1 public class S {

2

3 static public int intS;

4

5 public S () {

6 System.out.println("in S constructor");

7 }

8

9 static {

10 System.out.println("S:Static 1");

11 }

12

13 static {

14 System.out.println("S: Static 2");

15 }

16

17 public static void main(String args[]) {

18 System.out.println("new S() " + new S());

19 }

20 }

Source Code: [Src/6\_AddOn/S.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6_AddOn/S.java)

• SubclassOfS.java:

1 public class SubclassOfS extends S {

2

3 public int intS;

4

5 static {

6 System.out.println("SubclassOfS: Static 1");

7 }

8

9 public SubclassOfS () {

10 System.out.println("in SubclassOfS constructor");

11 }

12

13 public static void main(String args[]) {

14 System.out.println("In SubClass of S");

15 SubclassOfS aSubclassOfS = new SubclassOfS();

16 }

17 }

Source Code: [Src/6\_AddOn/SubclassOfS.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6_AddOn/SubclassOfS.java)

OUTPUT:-

S:Static 1

S: Static 2

SubclassOfS: Static 1

In SubClass of S

in S constructor

in SubclassOfS constructor

**Questions**

• Will it compile:

1 public class Yellow {

2 private String yellowPrivate = "yellowPrivate";

3

4 public static void main(String args[]) {

5 System.out.println(yellowPrivate);

6 }

7 }

NOPE - cant access non static variable from a static method without using object of class

Will it compile and if yes, what is the output: (eine neue variable wird erzeugt).

1 public class Coke {

2 private String cokePrivate = "cokePrivate";

3 private String s;

4 private static String cokePrivateS = "cokePrivateS";

5

6 public void m() {

7 cokePrivate = "java";

8 }

9

10 public void change(String cokePrivate) {

11 cokePrivate = "hello";

12 }

13 public void print() {

14 System.out.println("1. cokePrivate = " + cokePrivate );

15 System.out.println("2. cokePrivateS = " + cokePrivateS );

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

17 }

18 public static void main(String args[]) {

19 Coke aCoke = new Coke();

20 aCoke.m();

21 aCoke.print();

22 aCoke.change("t");

23 aCoke.print();

24 }

25 }

Source Code: [Src/6\_q/Coke.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6_q/Coke.java)

1. cokePrivate = java

2. cokePrivateS = cokePrivateS

----------------------------------

1. cokePrivate = java

2. cokePrivateS = cokePrivateS

----------------------------------

• Will it compile: (die antwort zu der frage kann nur verneint werden)

1 public class Red {

2 private String redPrivate = "redPrivate";

3 }

Source Code: [Src/6\_q/Red.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6_q/Red.java)

An other example:

1 public class Blue {

2 private String bluePrivate = "bluePrivate";

3

4 public boolean isLess(Red aRed) {

5 return bluePrivate == aRed.redPrivate;

6 }

7

8 public static void main(String args[]) {

9 Red aRed = new Red();

10 Blue aBlue = new Blue();

11 aBlue.isLess(aRed);

12 }

13 }

Source Code: [Src/6\_q/Blue.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6_q/Blue.java)

NOPE - java: redPrivate has private access in Red. The correct way to access a private variable from another class is with getter and setter methods. Otherwise, you should have made that variable public.

• Will it compile and if yes, what is the output:

1 public class H {

2 private String hPrivate = "hPrivate";

3 private static String hPrivateS = "hPrivateS";

4

5 public H(String hPrivate) {

6 this.hPrivate = hPrivate;

7 }

8

9 public void knete() {

10 this = this("RIT");

11 }

12

13 public void print(String tag) {

14 System.out.println(tag + "hPrivate = " + hPrivate );

15 }

16

17 public static void main(String args[]) {

18 H aH = new H();

19 aH.print("1.");

20 aH.knete();

21 aH.print("2.");

22 }

23 }

Source Code: [Src/6\_q/H.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6_q/H.java)

Execution:

H.java:10: Invalid left hand side of assignment.

this = this("RIT");

^

H.java:10: Only constructors can invoke constructors.

this = this("RIT");

^

H.java:18: No constructor matching H() found in class H.

H aH = new H();

Will it compile and if yes, what is the output:

1 public class Bauer {

2 private String bauerPrivate = "bauerPrivate";

3 private static String bauerPrivateS = "bauerPrivateS";

4

5 public Bauer() {

6 }

7

8 public Bauer(String bauerPrivate) {

9 this.bauerPrivate = bauerPrivate;

10 }

11

12 public void knete() {

13 this = new Bauer("RIT");

14 }

15

16 public static void main(String args[]) {

17 Bauer aBauer = new Bauer();

18 }

19 }

Source Code: [Src/6\_q/Bauer.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6_q/Bauer.java)

NOPE - can’t assign to “this”

Will it compile: (eins ist genug)

1 public class Bier {

2 private int bier;

3

4 public Bier() {

5 bier ++;

6 }

7

8 public void print() {

9 System.out.println("bier = " + bier );

10 }

11

12 public static void main(String args[]) {

13 Bier aBier = new Bier();

14 for ( int i = 0; i < 1000; i ++ )

15 aBier = new Bier();

16 aBier.print();

17 }

18 }

Source Code: [Src/6\_q/Bier.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/6_q/Bier.java)

OUTPUT:- bier = 1

**Upper Bounded Type Parameters**

• Upper Bound Type Parameters allow to restrict the types that can be used as type arguments

• <T extends Class>

A class that uses bound type parameters: Brick extends Cube extends Square extends Area

1 import java.util.LinkedList;

2 import java.util.List;

3 public class UpperB<T extends Cube> {

4

5 T theThing;

6

7 public UpperB(T theThing) {

8 this.theThing = theThing;

9 }

10 double area() {

11 return theThing.area();

12 }

13 public static void main(String[] args) {

14

15

16 // UpperB<Square> aSquare = new UpperB<Square>(new Square(12));

17 UpperB<Cube> aCube = new UpperB<Cube>(new Cube(12));

18 UpperB<Brick> aBrick = new UpperB<Brick>(new Brick(12, 24, 36));

19 // linked list op UpperB->Brick

20 List<UpperB<Brick>> aList = new LinkedList<UpperB<Brick>>();

21 // linked list op UpperB->Square

22 List<UpperB<Cube>> bList = new LinkedList<UpperB<Cube>>();

23 List<UpperB<? extends Cube>> cList = new LinkedList<UpperB<? extends Cube>>();

24 aList.add( new UpperB<Brick>(new Brick(12, 24, 36) ) );

25 bList.add( new UpperB<Cube>(new Cube(12) ) );

26 // cList.add( new UpperB<Square>(new Square(12) ) );

27 cList.add( new UpperB<Brick>(new Brick(12, 24, 36) ) );

28 /\*

29 cList.add( new UpperB<Square>(new Suare(12) ) );

30 UpperB.java:26: error: incompatible types: UpperB<Square> cannot be converted to UpperB<? extends Cube>

31 cList.add( new UpperB<Square>(new Suare(12) ) );

32 ^

33 \*/

34

35 }

36 }

Source Code: [Src/11\_W/UpperB.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/11_W/UpperB.java)

1 import java.util.LinkedList;

2 import java.util.List;

3 public class LowerB<T extends Cube> {

4

5 T theThing;

6

7 public LowerB(T theThing) {

8 this.theThing = theThing;

9 }

10 double area() {

11 return theThing.area();

12 }

13 public static void main(String[] args) {

14 LowerB<Square> aSquare = new LowerB<Square>(new Square(12));

15 LowerB<Cube> aCube = new LowerB<Cube>(new Cube(12));

16 LowerB<Brick> aBrick = new LowerB<Brick>(new Brick(12, 24, 36));

17 List<LowerB<Brick>> aList = new LinkedList<LowerB<Brick>>();

18 }

19 }

Source Code: [Src/11\_W/LowerB.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/11_W/LowerB.java)

Compilation:

LowerB.java:14: error: type argument Square is not within bounds of type-variable T

LowerB<Square> aSquare = new LowerB<Square>(new Square(12));

^

where T is a type-variable:

T extends Cube declared in class LowerB

LowerB.java:14: error: type argument Square is not within bounds of type-variable T

LowerB<Square> aSquare = new LowerB<Square>(new Square(12));

^

where T is a type-variable:

T extends Cube declared in class LowerB

2 errors

shell returned 1

**Multiple Bounds**

• A class can implement a class and multiple interfaces

public class AA extends A implements I1, I2 { ... }

• For multiple bounds we should a similar functionality:

public class D<T extends A & I2 & I1> { ... }

public static <T extends Comparable<T> & Iterator<T>> void method(T aT) {...}

• The most frequent use of multiple upper bounds is to implement two or more interfaces.

• Example:

public static <T extends Comparable<T> & Iterator<T>> void method(T aT) {...}

• A type variable with multiple bounds is a subtype of all the types listed in the bound.

• If one of the bounds is a class, it must be specified first. THe remaining ones must be interfaces.

1 class A {

2 public void a(){ }

3 }

4 interface I1 {

5 public void i1();

6 }

7 interface I2 {

8 public void i2();

9 }

10 class AA extends A implements I1, I2 {

11 public void i1() {}

12 public void i2() {}

13 }

14

15

16 class MultipleB2<T extends AA & I1 & I2> {

17 void test(T t) {

18 t.a();

19 t.i1();

20 t.i2();

21 }

22 public static void main(String[] args ) {

23 new MultipleB2<AA>().test(new AA() );

24 }

25 }

Source Code: [Src/11\_W/MultipleB2.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/11_W/MultipleB2.java)

**Generic Arrays**

Java’s arrays contain, at runtime, information about its component type. Since you don’t know what T is at runtime, you can’t create the array.

There for you can not create arrays of parameterized types

**Restrictions on Generics**

Stolen from: <https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>

• Cannot Instantiate Generic Types with Primitive Types

• Cannot Create Instances of Type Parameters

• Cannot Declare Static Fields Whose Types are Type Parameters

• Cannot Use Casts or instanceof With Parameterized Types

• Cannot Create Arrays of Parameterized Types

• Cannot Create, Catch, or Throw Objects of Parameterized Types

• Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

**Wildcards**

<https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html>

• In generic code, the question mark (?), called the wildcard, represents an unknown type.

• The wildcard can be used in a variety of situations:

— as the type of a parameter, field, or local variable;

— sometimes as a return type (though it is better programming practice to be more specific).

— The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a super-type.

**Upper Bounded Wildcards**

Stolen from above: <https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html>

Example: public static void process(List<? extends Number> )

All objects of classes which extend Number

• List<Integer>, List<Double>, and List<Number>;

• Integer|Double extends Number

Upper Bound Wildcard: "? extends E": Denotes a family of subtypes of type Type. This is the most useful wildcard

(bounded by its super class E) A method that works on lists of Vector and the subtype of Vector, such as Stack, *List<extends Vector>*

A method that works on lists of Integer and the super types of Integer, such as Integer, Number, and Object: *List<? super Integer>*

1 import java.util.\*;

2

3 public class UpperBound {

4 public static void main(String[] args) {

5 List<Double> listOfDoubles = new LinkedList<Double>();

6 List<Integer> listOfIntegers = new LinkedList<Integer>();

7

8 listOfDoubles.add( Double.valueOf(1.0) );

9 listOfIntegers.add( Integer.valueOf(2) );

10

11 System.out.println("sum of integer’s is: " + sum(listOfIntegers));

12 System.out.println("sum of double’s is: " + sum(listOfDoubles));

13 }

14

15 // instead of double?

16 private static double sum(List<? extends Number> list) {

17 double sum=0.0;

18 for (Number i: list)

19 sum += i.doubleValue();

20 return sum;

21 }

22 }

Source Code: [Src/11\_W/UpperBound.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/11_W/UpperBound.java)

Next

% java UpperBound

sum of integer’s is: 2.0

sum of double’s is: 1.0

**Unbounded Wildcards**

Unbound: "?": Denotes the set of all types or any

Example: public static void printList(List<?> list)

Can print a list of any type

Useful approach:

• a method that can be implemented using functionality provided in the Object class.

• code is using methods in the generic class that don’t depend on the type parameter.

• for ex: List.size() or List.clear().

• Class<?> is so often used because most of the methods in Class<T> do not depend on T.

1 import java.util.\*;

2

3 public class Unbound {

4

5 public static void printList(List<?> list) {

6 for (Object elem: list)

7 System.out.print(elem + " ");

8 System.out.println();

9 }

10

11 public static void main(String args[]) {

12 List<Integer> listOfIntegers = Arrays.asList(1, 2, 3);

13 List<String> listOfStrings = Arrays.asList("a", "b", "c");

14 printList(listOfIntegers);

15 printList(listOfStrings);

16 }

17 }

18

Source Code: [Src/11\_W/Unbound.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/11_W/Unbound.java)

Output:

% java Unbound

1 2 3

a b c

**Lower Bounded Wildcards**

A lower bounded wildcard is expressed using the wildcard character (’?’), following by the super keyword, followed by its lower bound:

• <? super Cube

Brick extends Cube extends Square extends Area

a method which can only print cubes and brick, but not squares

1 import java.util.LinkedList;

2 import java.util.List;

3 import java.util.Date;

4 import java.sql.Time;

5

6 class LowerBound {

7 public static void main(String[] args) {

8 List<Cube> listOfCube = new LinkedList<Cube>();

9 List<Brick> listOfBrick = new LinkedList<Brick>();

10 List<Square> listOfSquare = new LinkedList<Square>();

11

12 listOfCube.add( new Cube(1) );

13 listOfBrick.add( new Brick(1, 2, 3) );

14 listOfSquare.add( new Square(1) );

15 /\*

16 print(listOfBrick);

17 LowerBound.java:16: error: incompatible types: List<Brick> cannot be converted to List<? super Cube>

18 print(listOfBrick);

19 ^

20 Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output

21 1 error

22 \*/

23 print(listOfCube);

24 print(listOfSquare);

25 }

26

27 public static void print(List<? super Cube> list) {

28 System.out.println(list);

29 }

30 }

Source Code: [Src/11\_W/LowerBound.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/11_W/LowerBound.java)

[Date()](https://docs.oracle.com/javase/8/docs/api/java/util/Date.html)

% java LowerBound

[Cube@7e9e5f8a]

[Square@8bcc55f]

**Summary**

• Upper bound: public static void process(List<? extends Number> )

Number class and sub classes of number, Integer, Double

• Unbound: public static void printList(List<?> list) -

List of all types

• Lower Bounded public static void addNumbers(List<? super Integer> list)

Integer and super classes of Integer, Integer, Numbers, Object

**8.24. Bound Types vs Wildcards**

• Bounded type parameters can have multiple bounds

• Wildcards can have a lower or upper bound. There is no lower bound for type parameters

• Bounded type parameters allow to express dependencies among the types of one or more arguments in a method and/or the return type

**Bound Wildcards**

•

public abstract class Shape {

public abstract void draw(Canvas c);

}

public class Circle extends Shape {

private int x, y, radius;

public void draw(Canvas c) { ... } }

public class Rectangle extends Shape {

private int x, y, width, height;

public void draw(Canvas c) { ... } }

• These classes can be drawn on a canvas:

public class Canvas { public void draw(Shape s) {

s.draw(this);

}

}

• Any drawing will typically contain a number of shapes.

• Assuming that they are represented as a list, it would be convenient to have a method in Canvas that draws them all:

public void drawAll(List<Shape> shapes) {

for (Shape s: shapes) {

s.draw(this);

}

• What we really want is:

public void drawAll(List<? extends Shape> shapes) { ... }

• *class C<T extends Figure> {..}*

— T must descend from Figure.

— Allows Figure’s features to be referenced.

— Keyword extends is even used for interfaces!

• Example:

public void drawAll(List<? extends Shape> shapes) { ... }

• List<? extends Shape> is an example of a bounded wildcard

• However, in this case, we know that this unknown type is in fact a subtype of Shape 1. Shape is the upper bound of the wildcard.

• This defines upper bounds

• The type Collection<? super String> is a super type of any Collection where T is a super type of String

• This collection can store Strings and Objects

**VERY IMPORTANT LESSON**

Brick extends Cube extends Square extends Area

I.e. Brick is the child of Cube: which is child of Square: which is Child of Area

**NUMBER 1:-**

Public void functionName(? extends Brick){

**// this function accepts brick and its children - none in this case**

…

}

**NUMBER 2:-**

Public void functionName(? super Cube){

**// this function accepts Cube and its parents - Square and Area**

…

}

HOWEVER,

**NUMBER 3:-**

List<? super Brick> aList = new ArrayList();

// this line above can only accept Brick and its children, and not its parents

**NUMBER 4:-**

List<? extends Brick> aList = new ArrayList();

// this line above can only accept null, and not its children of brick, as at run-time the system wouldn’t be aware what type of generic is actually being used, as ? can be anything, this list instance is read-only and therefore can’t add anything but null.

**Exceptions**

See also: [Exceptions](https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Exception.html)

When a Java program violates the semantic constraints of the Java language, a Java Virtual Machine signals this error to the program as an exception. An example of such a violation is an attempt to index outside the bounds of an array.

Java specifies that an exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred.

Java programs can also throw exceptions explicitly, using throw statement.

The Java language checks, at compile time, that a Java program contains handlers for checked exceptions, by analyzing which checked exceptions can result from execution of a method or constructor.

An Error indicates are serious, most likely not recoverable, situation.

**Runtime Exceptions**

• [Runtime Exceptions](https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html)

• unchecked exceptions classes are the class RuntimeException and its subclasses

• class Error and its subclasses

Example:

1 public class RunTime {

2

3 private void divide(int a, int b) {

4 int result = a / b;

5 }

6 private void callDivider(int a, int b) {

7 divide(a, b);

8 }

9

10 public static void main(String[] args) {

11 int a = (int) Math.random();

12 int b = 1;

13 new RunTime().callDivider(a, b);

14 new RunTime().callDivider(b, a);

15 }

16 }

Source Code: [Src/7/RunTime.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/7/RunTime.java)

**Compile Time Exceptions**

• All others

• The list is long

Example:

1 import java.io.FileNotFoundException;

2 import java.util.Scanner;

3 import java.io.File;

4

5 public class CompileTime {

6

7 private void openFile (String filenName ) throws FileNotFoundException {

8 new Scanner(new File(filenName) );

9 }

10 private void callOpenFile() {

11 try {

12 openFile("thisFileDoesNotExist.txt");

13 } catch ( FileNotFoundException e ) {

14 System.out.println("getCasue: " + e.getCause());

15 e.printStackTrace();

16 }

17 }

18

19 public static void main(String[] args) {

20 new CompileTime().callOpenFile();

21 }

22 }

**Runtime Exceptions--The Controversy**

Copied from: [read here ...](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html)

• Although Java requires that methods catch or specify checked exceptions, they do not have to catch or specify runtime exceptions, that is, exceptions that occur within the Java runtime system.

• Because catching or specifying an exception is extra work, programmers may be tempted to write code that throws only runtime exceptions and therefore doesn’t have to catch or specify them.

• This is "exception abuse" and is not recommended.

**Finally I**

• The finally block will be always executed, regardless of what happens in the try block, or in the catch blocks.

• System.exit however will terminate the JVM.

This provides a place where you can put statements which will be always executed.

finally ( )

{

statement sequence

}

First example:

1

2 import java.util.Vector;

3

4 public class Finally\_0 {

5

6 private void test\_1() {

7 try {

8 String aString = "a:";

9 aString = null;

10 aString.length();

11 } catch (NullPointerException e) {

12 System.out.println("NullPointerException caught");

13 } finally {

14 System.out.println("finally 1");

15 }

16 }

17 private void test\_2() {

18 try {

19 String aString = "a:";

20 aString = null;

21 } catch (Exception e) {

22 System.out.println(e);

23 } finally {

24 System.out.println("finally 2");

25 }

26 }

27 private void test\_3() {

28 try {

29 String aString = "a:";

30 aString = null;

31 aString.length();

32 } catch (NullPointerException e) {

33 System.out.println("NullPointerException caught");

34 System.exit(0);

35 } finally {

36 System.out.println("finally 1");

37 }

38 }

39 public static void main(String[] args) {

40 new Finally\_0().test\_1();

41 new Finally\_0().test\_2();

42 new Finally\_0().test\_3();

43 }

44 }

Source Code: [Src/7/Finally\_0.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/7/Finally_0.java)

Output:

finally 1

finally 2

NullPointerException caught

**Finally and Return and Throw Exceptions**

1

2 public class Finally\_1 {

3

4 private int test\_1() {

5 try {

6 String aString = "a:";

7 aString = null;

8 aString.length();

9 } catch (NullPointerException e) {

10 System.out.println("catch ");

11 return 0;

12 } finally {

13 System.out.println("finally");

14 return 1;

15 }

16 }

17 private int test\_2() {

18 try {

19 String aString = "a:";

20 aString = null;

21 aString.length();

22 } catch (Exception e) {

23 throw new Exception("3");

24 } finally {

25 System.out.println("finally");

26 return 1;

27 }

28 }

29 public static void main(String[] args) {

30 System.out.println("Rvalue = " + new Finally\_1().test\_1());

31 System.out.println("Rvalue = " + new Finally\_1().test\_2());

32 }

33 }

Source Code: [Src/7/Finally\_1.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/7/Finally_1.java)

Output:

catch

finally

Rvalue = 1

finally

Rvalue = 1

**Exception: Ordering, Hierarchy, and Finally**

[see here](https://docs.oracle.com/javase/7/docs/api/java/lang/ArrayIndexOutOfBoundsException.html)

java.lang.Object

java.lang.Throwable

java.lang.Exception

java.lang.RuntimeException

java.lang.IndexOutOfBoundsException

java.lang.ArrayIndexOutOfBoundsException

Catch blocks order from lowest sub class upwards

1

2 public class CatchOrder {

3

4 private void anExeption1() {

5 int[] anArray = new int[1];

6 try {

7 System.out.println("Before execption causing instruction");

8 anArray[2] = 1; // exception will be thrown

9 System.out.println("After execption causing instruction");

10 /\*

11 } catch (Exception e) {

12 CatchOrder.java:10: error: exception ArrayIndexOutOfBoundsException has already been caught

13 } catch (ArrayIndexOutOfBoundsException e) {

14 ^

15 CatchOrder.java:11: error: exception IndexOutOfBoundsException has already been caught

16 } catch (IndexOutOfBoundsException e) {

17 ^

18 2 errors

19 \*/

20 } catch (ArrayIndexOutOfBoundsException e) {

21 e.printStackTrace();

22 } catch (IndexOutOfBoundsException e) {

23 }

24 }

25

26 public static void main(String[] args) {

27 try {

28 new CatchOrder().anExeption1();

29 } catch ( Exception e ) {

30 e.printStackTrace();

31 }

32 }

33 }

Source Code: [Src/7/CatchOrder.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/7/CatchOrder.java)

Finally - return and throw

finally ’overwrites’ an exception thrown in a catch block

1

2 public class CatchOrderFinally {

3

4 private void anExeption1() {

5 int[] anArray = new int[1];

6 String s = "test";

7 try {

8 anArray[2] = 1;

9 } catch (ArrayIndexOutOfBoundsException e) {

10 System.out.println("Caught ArrayIndexOutOfBoundsException in anExeption1()");

11 s = null;

12 s.length();

13 } finally {

14 System.out.println("finally");

15 }

16 }

17 private void anExeption2() {

18 int[] anArray = new int[1];

19 String s = "test";

20 try {

21 anArray[2] = 1;

22 } catch (ArrayIndexOutOfBoundsException e) {

23 System.out.println("Caught ArrayIndexOutOfBoundsException in anExeption2()");

24 s = null;

25 s.length(); // this does not end in main with a nullPointerEx.

26 } finally {

27 System.out.println("finally");

28 anArray[2] = 1;

29 }

30 }

31 /\*

32 Caught ArrayIndexOutOfBoundsException in anExeption1()

33 finally

34 Caught NullPointerException in main

35 Caught ArrayIndexOutOfBoundsException in anExeption2()

36 finally

37 Caught ArrayIndexOutOfBoundsException in main

38 \*/

39

40 public static void main(String[] args) {

41 try {

42 new CatchOrderFinally().anExeption1();

43 } catch ( Exception e ) {

44 System.out.println("Caught NullPointerException in main");

45 }

46 try {

47 new CatchOrderFinally().anExeption2();

48 } catch ( Exception e ) {

49 System.out.println("Caught ArrayIndexOutOfBoundsException in main");

50 }

51 }

52 }

Source Code: [Src/7/CatchOrderFinally.java](https://www.cs.rit.edu/~hpb/Lectures/2211/605/Src/7/CatchOrderFinally.java)

**THREADS**

Difference between start() and run()

| start() | run() |
| --- | --- |
| Creates a new thread and the run() method is executed on the newly created thread. | No new thread is created and the run() method is executed on the calling thread itself. |
| Can’t be invoked more than one time otherwise throws *java.lang.IllegalStateException* | Multiple invocation is possible |
| Defined in *java.lang.Thread* class. | Defined in *java.lang.Runnable* interface and must be overridden in the implementing class. |

1. Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start() creates a new Thread that executes the code written in run(). Calling run() directly doesn’t create and start a new Thread, it will run in the same thread. To start a new line of execution, call start() on the thread.

**Thread\_0**

1

2 public class Thread\_0 extends Thread {

3 private String info;

4 static Object o = new Object();

5

6 public Thread\_0 (String info) {

7 this.info = info;

8 }

9

10 public void run () {

11 System.err.println(info + " ---> ");

12 try { sleep(1000); } catch ( InterruptedException e ) { }

13 System.err.println(info + " <--- ");

14 }

15

16 public static void main (String args []) {

17 Thread\_0 aT4\_0 = new Thread\_0("first");

18 Thread\_0 aT4\_1 = new Thread\_0("second");

19

20 aT4\_0.start();

21 aT4\_1.start();

22 }

23 }

Competing Threads

Threads can obtain exclusive access to an object if all competing threads use a synchronized statement or call a method with synchronized attribute. Class methods monitor the class description, other methods monitor their receiver, the statement monitors the indicated value. The attribute synchronized precedes the result type.

Adding or deleting a synchronized modifier of a method does not break compatibility with existing binaries.

•

the execution of methods can be synchronized.

public synchronized method( ...) { ... }

public static synchronized method( ...) { ... }

•

synchronized statements which allow access to an associated object.

synchronized(aObj) { ... }

public synchronized method( ...) { ... }

is aequivalent to

public method( ...) {

synchronized ( this ) {

}

}

Note: Interface methods can’t be native, static, synchronized, final, private, or protected

import java.util.\*;

public class ClassT extends Thread {

private String info;

private Vector aVector;

public ClassT (String info, Vector aVector) {

this.info = info;

this.aVector = aVector;

}

static synchronized void staticInProtected1(String s) {

System.err.println(s + ": ---->");

try {

sleep(1000);

}

catch ( InterruptedException e ) {

System.err.println("Interrupted!");

}

staticInProtected2(s);

System.err.println(s + ": <----");

}

static synchronized void staticInProtected2(String s) {

System.err.println(s + ": ====>");

try {

sleep(1000);

}

catch ( InterruptedException e ) {

System.err.println("Interrupted!");

}

System.err.println(s + ": ====>");

}

public void run () {

staticInProtected1(info);

}

public static void main (String args []) {

Vector aVector = new Vector();

ClassT aClassT\_0 = new ClassT("first", aVector);

ClassT aClassT\_1 = new ClassT("second", aVector);

ClassT.staticInProtected1("main");

aClassT\_0.start();

aClassT\_1.start();

aClassT\_0.staticInProtected1("aClassT\_0");

aClassT\_1.staticInProtected1("aClassT\_1");

}

}

When in “wait”, the thread gives up the lock if in synhronized block but not when in sleep in a synchronized block.