Java Programming 2

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Java 281
Spring 2017

```java
class Lecture7 {

        // Objects and Classes

}

// Key words:
class, new, this, static, null, extends, super, abstract, final,
     interface, implements, protected
```

Observations for Real Objects

- Look around.
- We can easily find many examples for real-world objects.
  - For example, a person and his/her bottle of water.
- Real-world objects all have states and behaviors.
  - What possible states can the object be in?
  - What possible behaviors can the object perform on the states?
- Identifying these states and behaviors for real-world objects is a great way to begin thinking in object-oriented programming.
- From now, OO is a shorthand for "object-oriented."

Software Objects

- An object keeps its states in fields and exposes its behaviors through methods.
- Plus, internal states are hidden and the interactions to the object are only performed through an object's methods.
- This is so-call encapsulation, which is one of OO features.
- Note that the other OO features are inheritance and polymorphism, which we will see later.

Classes

- We often find many individual objects all of the same kind.
  - For example, each bicycle was built from the same blueprint so that each contains the same components.
- In OO terms, we say that your bicycle is an instance of the class of objects known as **Bicycle**.
- A class is the blueprint to create class instances which are runtime objects.
- Classes are the building blocks of Java applications.

Example: Points in 2D Coordinate

```java
class Point {
    double x, y; // fields: data member
}
```

```java
public class PointDemo {
    public static void main(String[] args) {
         // now create a new instance of Point
        Point p1 = new Point();
        p1.x = 1;
        p1.y = 2;
        System.out.printf("(%d, %d)\n", p1.x, p1.y);

        // create another instance of Point
        Point p2 = new Point();
        p2.x = 3;
        p2.y = 4;
        System.out.printf("(%d, %d)\n", p2.x, p2.y);
    }
}
```

Class Definition

- First, give a class name with the first letter capitalized, by convention.
- The class body, surrounded by balanced braces $\{\}$, contains data members (fields) and function members (methods) for objects.

Data Members

- The fields are the states of the object.
- The field may have an access modifier, say public and private.
  - public: accessible from all classes
  - private: accessible only within its own class
- You can decide if these fields are accessible!
- In practice, all fields should be declared private.
- However, this private modifier does not quarantine any security.[1]
  - What private is good for maintainability and modularity.[2]

[1] Thanks to a lively discussion on January 23, 2017.

[2] Read `http://stackoverflow.com/questions/9201603/are-private-members-really-more-secure-in-java`.

Function Members

- As said, the fields are hidden.
- So we may need accessors and mutators if necessary.
  - Accessors: return the state of the object
  - Mutators: set the state of the object
- For example, **getX**() and **getY**() are accessors, and **setPoint**(double, double) is one mutator in the class **Point**.

Example: Point (Encapsulated)

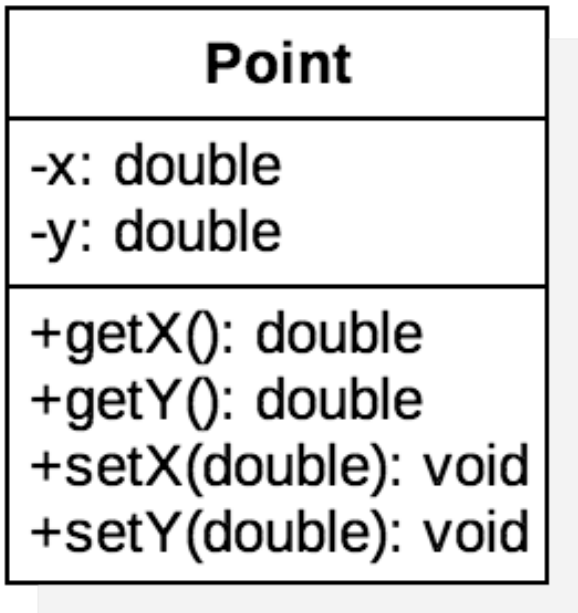
```
1  class Point {
2      private double x;
3      private double y;
4
5      double getX() { return x; }
6      double getY() { return y; }
7
8      void setX(double a) { x = a; }
9      void setY(double a) { y = a; }
10     void setPoint(double a, double b) {
11         x = a;
12         y = b;
13     }
14 }
```

Unified Modeling Language[3]

- Unified Modeling Language (UML) is a tool for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.
- Free software:
  - `http://staruml.io/` (available for all platforms)

[3] See `http://www.tutorialspoint.com/uml/` and `http://www.mitchellsoftwareengineering.com/IntroToUML.pdf`.

Example: Class Diagram for Point



- Modifiers can be placed before the fields and the methods:
  - $+$ for public
  - $-$ for private

Constructors

- A constructor is called by the new operator.
- A constructor acts like other methods.
- However, its names should be identical to the name of the class and it has no return type.
- A class may have several constructors if needed.
  - Constructors can be overloaded.
- Note that the constructors are used only during the objection creation.
  - Constructors cannot be invoked by any object.
- If you don't define any explicit constructor, Java assumes a default constructor for your class.
- Moreover, adding any explicit constructor disables the default constructor.

Parameterized Constructors

- You can provide specific information to the parameterized constructor during the object creation.
- For example,

```
1  class Point {
2      ...
3
4      Point() {} // restore a default constructor;
5
6      // parameterized constructor
7      Point(double a, double b) {
8          x = a;
9          y = b;
10     }
11     ...
12 }
```

Self-reference

- You can refer to any (instance) member of the current object within methods and constructors by using this.
- The most common reason for using the this keyword is because a field is shadowed by method parameters.
- You can also use this to call another constructor in the same class by invoking this().

Example: Point (Revisited)

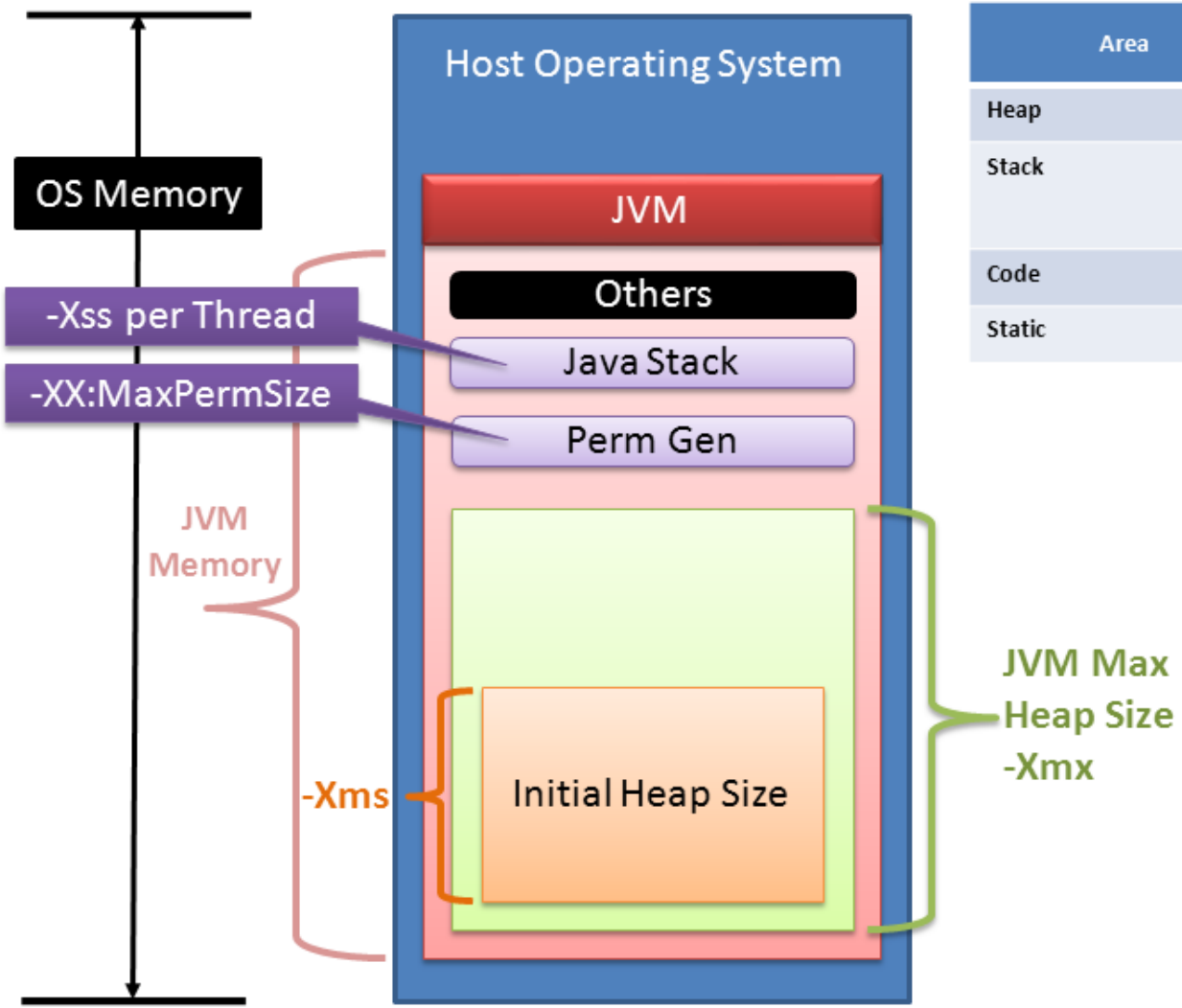
```
1 class Point {
2     ...
3     Point(int x, int y) {
4         this.x = x;
5         this.y = y;
6     }
7     ...
8 }
```

- Note that the this operator cannot be used in static methods.

Instance Members and Static Members

- You may notice that, until now, all members are declared w/o static.
- It means that each object has its own values with behaviors.
- The aforesaid members are called instance members.
- Note that these instance members are available only after the object is created.

Memory used by JVM



Area	Contains
Heap	Objects
Stack	- Methods - Local Variables - Reference Variables
Code	Byte Code
Static	Static Methods and Data

-Xmx : The JVM max Heap Size

-Xss : The Java Thread Stack Size, the default is OS and JVM dependent, and it can range 256k-to-1MB.The default should be tuned down to a range that doesn't cause StackOverflow. I often use 128k-192k. Since the default –Xss is high, tuning it down can help save on memory used and given back to the Guest OS.

Perm Size is an area additional to the –Xmx (Max Heap) value and is not GC'ed because it contains class-level information.

"other mem" is additional mem required for NIO buffers, JIT code cache, classloaders, Socket Buffers (receive/send), JNI, GC internal info

JVM Memory = JVM Max Heap (-Xmx value) + JVM Perm Size (-XX:MaxPermSize) + NumberOfConcurrentThreads * (-Xss value) + "other mem"

Static Members

- The static members belong to the class[4], and are shared between the instance objects.
- In other word, there is only one copy of the static members, no matter how many objects of the class are created.
- They are ready once the class is loaded.
- They can be invoked directly by the class name without using any instance.
- For example, **Math.random**().

[4] Aka class members.

- A static method can access other static members. (Trivial.)
- However, static methods cannot access to instance members directly. (Why?)
- For example,

```
1  ...
2      double getDistanceFrom(Point p) {
3          return Math.sqrt(Math.pow(this.x - p.x, 2) + Math.pow(
               this.y - p.y, 2));
4      }
5
6      static double distanceBetween(Point p1, Point p2) {
7          // You cannot access to x and y directly!
8          return Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.
               y - p2.y, 2));
9      }
10 ...
```

Example: Count of Points

```
1  class Point {
2      ...
3      private static int numOfPoint = 0;
4
5      Point() {
6          numOfPoint++;
7      }
8
9      Point(int x, int y) {
10         this(); // calling the constructor with no input
               argument; should be placed in the first line in the
                constructor
11         this.x = x;
12         this.y = y;
13     }
14     ...
15 }
```

Exercise: Singleton[5]

- In some situations, you may create the only instance of the class.

```
1  class Singleton {
2
3      // Will be ready as soon as the class is loaded.
4      private static Singleton instance = new Singleton();
5
6      // Do now allow to invoke the constructor by other classes.
7      private Singleton() {}
8
9      // Only way to obtain the singleton from the outside world.
10     public static Singleton getSingleton() {
11         return instance;
12     }
13 }
```

[5] See any textbook for design patterns.

Garbage Collection (GC)[6]

- Java handles deallocation automatically.
- Automatic GC is the process of looking at the heap memory, identifying whether or not the objects are in use, and deleting the unreferenced objects.
- An object is said to be unreferenced if the object is no longer referenced by any part of your program.
  - Simply assign null to the reference to make the object unreferenced.
- So the memory used by these objects can be reclaimed.

[6] http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

finalize()

- The method **finalize**() conducts a specific task that will be executed right before the object is reclaimed by GC.
- The **finalize**() method can be only invoked prior to GC.
- In practice, it must not rely on the **finalize**() method for normal operations. (Why?)

Example

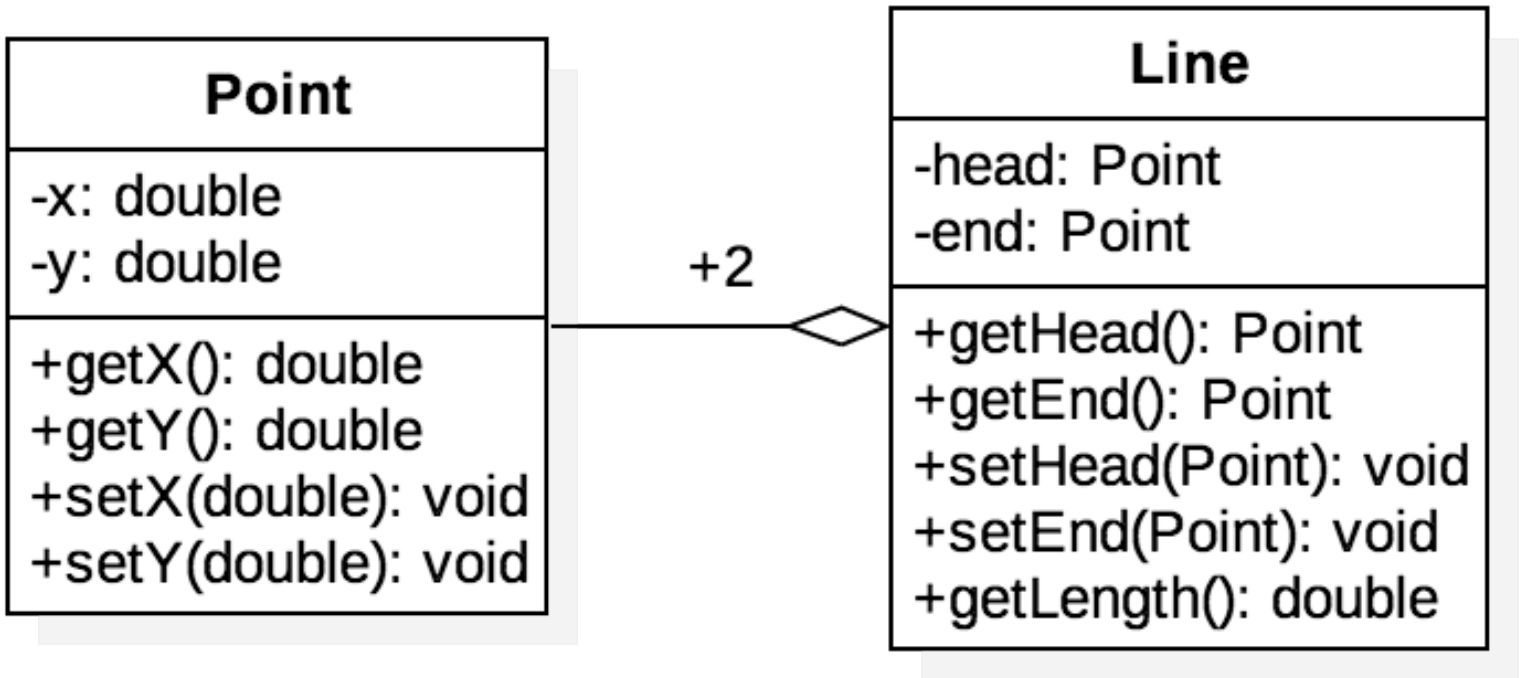
```
1  public class FinalizeDemo {
2      private static int numOfObjKilled = 0;
3
4      public void finalize() {
5          numOfObjKilled++;
6      }
7
8      public static void main(String[] args) {
9          double n = 1e7;
10         for (int i = 1; i <= n; i++)
11             new FinalizeDemo(); // lots of unreferenced objects
12         System.out.println(numOfObjKilled);
13     }
14 }
```

- You may try different number for instance creation.
- The number of the objects reclaimed by GC is uncertain.

HAS-A Relationship

- Association is a weak relationship where all objects have their own lifetime and there is no ownership.
  - For example, teacher $\leftrightarrow$ student; doctor $\leftrightarrow$ patient.
- If A uses B, then it is an aggregation, stating that B exists independently from A.
  - For example, knight $\leftrightarrow$ sword; company $\leftrightarrow$ employee.
- If A owns B, then it is a composition, meaning that B has no meaning or purpose in the system without A.
  - For example, house $\leftrightarrow$ room.

Example: Lines



- +2: two **Point** objects used in one **Line** object.

More Examples

- **Circle**, **Triangle**, and **Polygon**.
- **Book** with **Author**s.
- **Lecturer** and **Student**s in the classroom.
- **Zoo** with many creatures, say **Dog**, **Cat**, and **Bird**.
- **Channel**s played on **TV**.
- More.

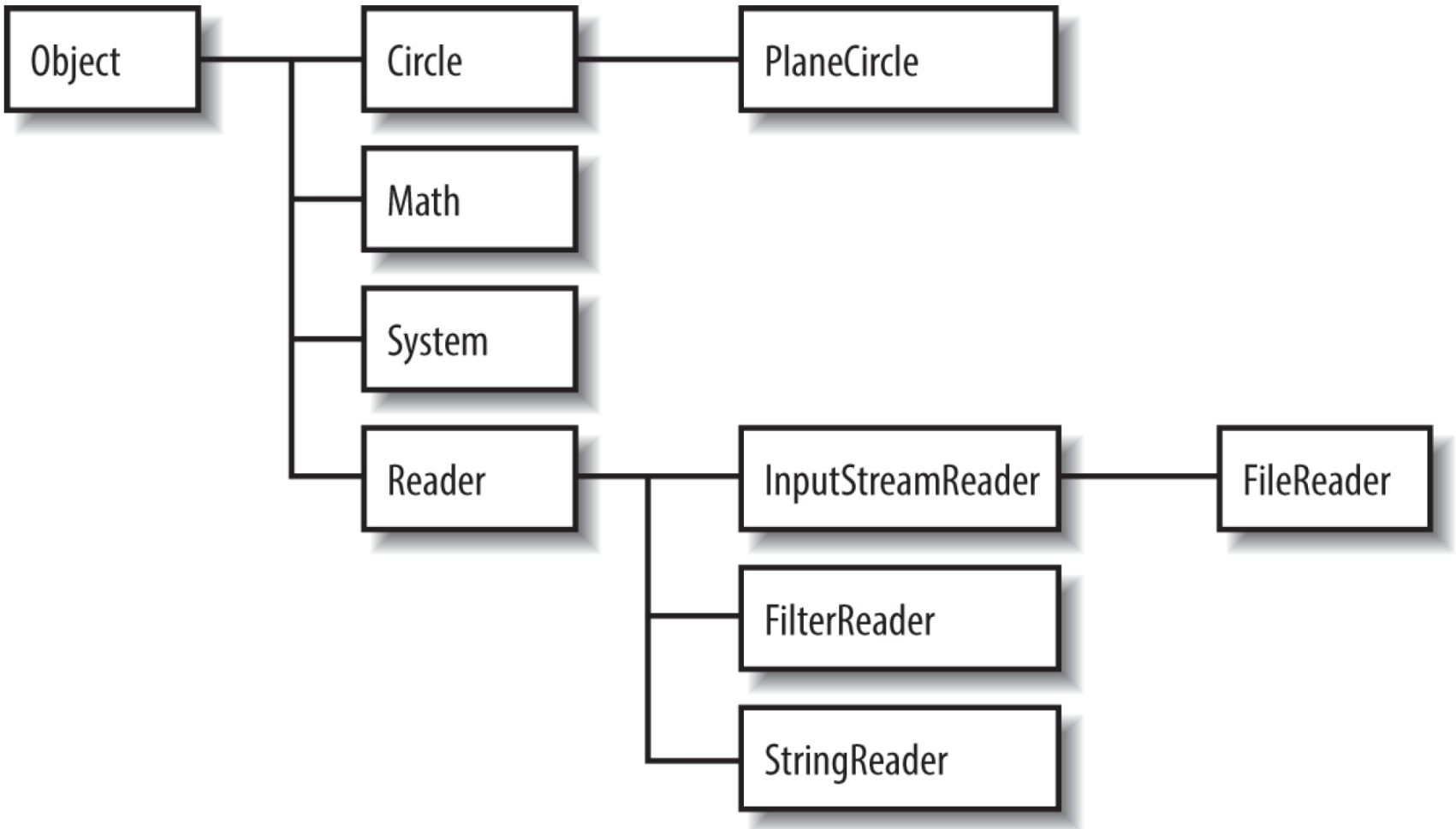
More Relationships Between Classes

- Inheritance: passing down states and behaviors from the parents to their children
- Interfaces: grouping the methods, which belongs to some classes, as an interface to the outside world
- Packages: grouping related types, providing access protection and name space management

First IS-A Relationship

- OOP allows classes to inherit commonly used states and behaviors from previously defined classes.
- This is called inheritance.
- Furthermore, the classes exist in some hierarchy.
- A class can be declared as a subclass of some class, which is called the superclass, by using the extends keyword.
- Hence, we can say that a subclass specializes its superclass.
- Equivalently, one subclass is a special case of the superclass.
  - For example, human and dog are two specific types of animals.
- Note that a class can extend only one other class, while each superclass has the potential for an unlimited number of subclasses.

Class Hierarchy[7]



[7] See Fig. 3-1 in p. 113 of Evans and Flanagan.

Example

```
1  class Animal {
2      String name;
3      int weight;
4
5      Animal(String s, int w) { name = s; weight = w; }
6
7      void eat() { weight += 1; }
8      void exercise() { weight -= 1; }
9  }
10
11 class Human extends Animal {
12     Human(String s, int w) { super(s, w); }
13     void writeCode() { System.out.println("Write codes..."); }
14 }
15
16 class Dog extends Animal {
17     Dog(String s, int w) { super(s, w); }
18     void watchDoor() { System.out.println("Watch my door..."); }
19 }
```

super

- Recall that the keyword this is used to refer to the object itself.
- You can use the keyword super to refer to (non-private) members of the superclass.
- Note that super() can be used to invoke the constructor of its superclass, just similar to this().

Constructor Chaining

- As the constructor is invoked, the constructor of its superclass is invoked accordingly.
- You might think that there will be a whole chain of constructors called, all the way back to the constructor of the class **Object**, the topmost class in Java.
- So every class is an immediate or a distant subclass of **Object**.
- Recall that the method **finalize**() and **toString**() are inherited from **Object**.
  - **toString**(): return a string which can be any information stored in the object.

Example

```
1  class A {
2      A() { System.out.println("A is creating..."); }
3  }
4
5  class B extends A {
6      B() { System.out.println("B is creating..."); }
7      public String toString() {
8          return "This is inherited from Object."
9      }
10 }
11
12 public class ConstructorChainingDemo {
13     public static void main(String[] args) {
14         B b = new B();
15         System.out.println(b);
16     }
17 }
```

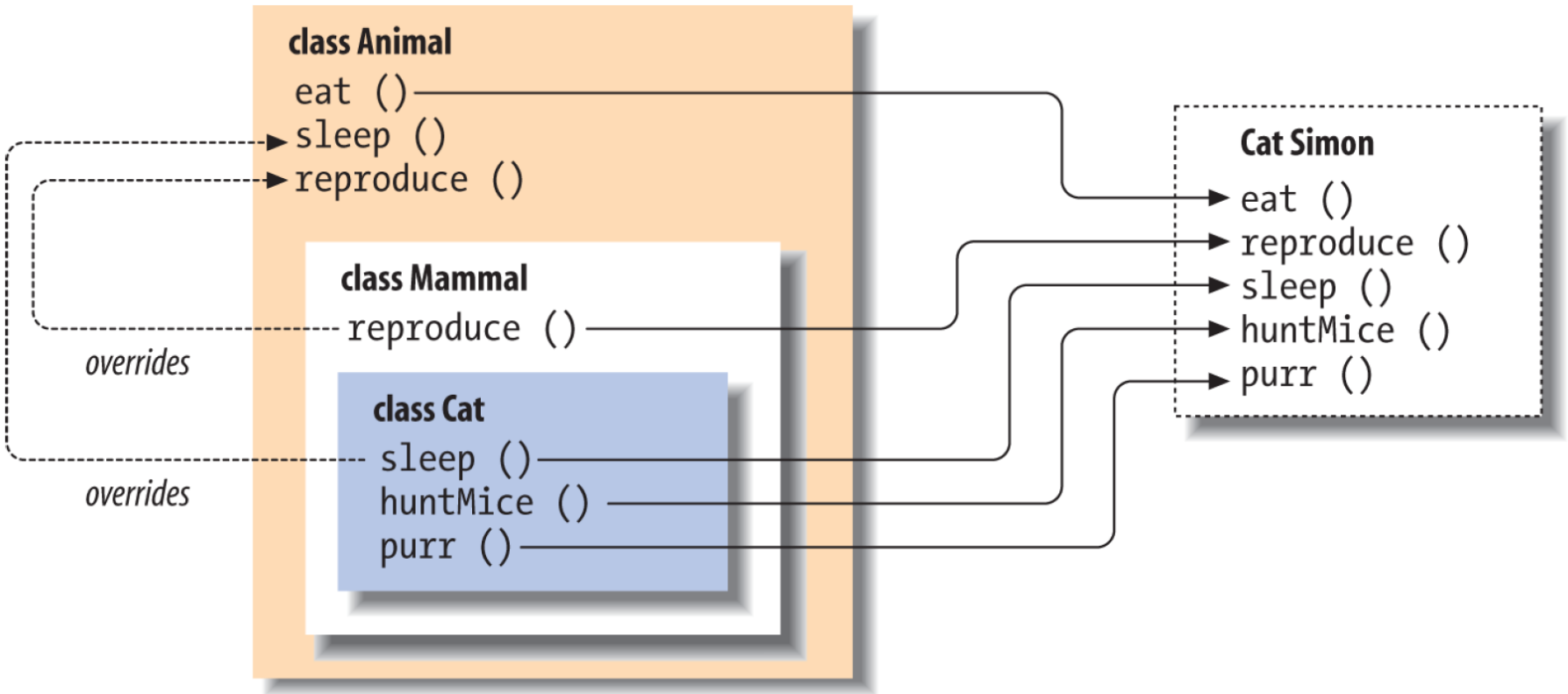
- The **println**() method (and similar methods) can take an object as input, and invoke **toString**() method implicitly.

Method Overriding

- The subclass is allowed to change the behavior inherited from its superclass, if needed.
- If one defines an instance method with its method name, parameters, and its return type, all identical to the previously defined method in its superclass, then this newly defined method overrides the one in the superclass.[8]
  - Compared to overridden methods, method overloading occurs only in the same class.
- Note that you can invoke the overridden method through the use of the keyword super.

[8] The static methods do not follow this rule.

Example


class Animal
eat ()
sleep ()
reproduce ()
class Mammal
reproduce ()
class Cat
sleep ()
huntMice ()
purr ()
overrides
overrides
Cat Simon
eat ()
reproduce ()
sleep ()
huntMice ()
purr ()

Binding

- Association of the method definition to the method call is known as binding.
- The binding which can be resolved at the compilation time is known as static binding or early binding.
  - They are the static, private or final methods.[9]
- If the compiler is not able to resolve the binding, such binding is known as dynamic binding or late binding.
  - For example, method overriding.
- When there are multiple implementations of the method in the inheritance hierarchy, the one in the "most derived" class (the furthest down the hierarchy) always overrides the others, even if we refer to the object through a reference variable of the superclass type.[10]

[9] We will see the final keyword soon.

[10] An overridden method in Java acts like a virtual function in C++.

Polymorphism[11]

- The word polymorphism literally means "many forms."
- Java allows 4 types of polymorphism:
  - coercion (casting)
  - ad hoc polymorphism (overloading)
  - subtype polymorphism
  - parametric polymorphism (generics)
- Modeling polymorphism in a programming language lets you create a uniform interface to different kinds of operands, arguments, and objects.

[11] Read `http://www.javaworld.com/article/3033445/learn-java/java-101-polymorphism-in-java.html`.

Subtype Polymorphism

- For convenience, let **U** be a subtype of **T**.
- Liskov Substitution Principle states that **T**-type objects may be replaced with **U**-type objects without altering any of the desirable properties of **T** (correctness, task performed, etc.).[12,13]

[12] See https://en.wikipedia.org/wiki/Liskov_substitution_principle.

[13] Also see https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

Casting

- Upcasting (widening conversion) is to cast the **U** object to the **T** variable.

```
1 T t = new U();
```

- Downcasting (narrow conversion) is to cast the **T** variable to a **U** variable.

```
1 U u = (U) t; // t is T variable reference to a U object.
```

- Upcasting is always allowed, but downcasting is allowed only when a **U** object is passed to the **U**-type variable.
- This involves type compatibility by JVM during program execution.

instanceof

- The operator instanceof allows us to test whether or not a reference variable is compatible to the object.
- If not compatible, then JVM will throw an exception **ClassCastException**.[14]

[14] We will see the exceptions later.

Example

```
1  class T {}
2  class U extends T {}
3
4  public class InstanceofDemo {
5      public static void main(String[] args) {
6          T t1 = new T();
7
8          System.out.println(t1 instanceof U); // output false
9          System.out.println(t1 instanceof T); // output true
10
11         T t2 = new U(); // upcasting
12
13         System.out.println(t2 instanceof U); // output true
14         System.out.println(t2 instanceof T); // output true
15
16         U u = (U) t2; // downcasting; this is ok.
17
18         u = (U) new T(); // pass the compilation; fail during
               execution!
19     }
20 }
```

Abstraction by Method Overriding and Polymorphism

- JVM invokes the appropriate method for the current object by looking up from the bottom of the class hierarchy to the top.
- These methods are also called virtual methods.
- This mechanism preserves the behaviors of the objects and the super-type variables play the role of placeholders.
- We manipulate objects in an abstract level; we don't need to know the details when we use them.

Example

- Imagine that we have a zoo with some animals.

```
1  class Animal {
2      void speak() {}
3  }
4  class Dog extends Animal {
5      void speak() { System.out.println("woof"); }
6  }
7  class Cat extends Animal {
8      void speak() { System.out.println("meow"); }
9  }
10 class Bird extends Animal {
11     void speak() { System.out.println("tweet"); }
12 }
13
14 public class PolymorphismDemo {
15     public static void main(String[] args) {
16         Animal[] zoo = {new Dog(), new Cat(), new Bird()};
17         for (Animal a: zoo) a.speak();
18     }
19 }
```

The final Keyword

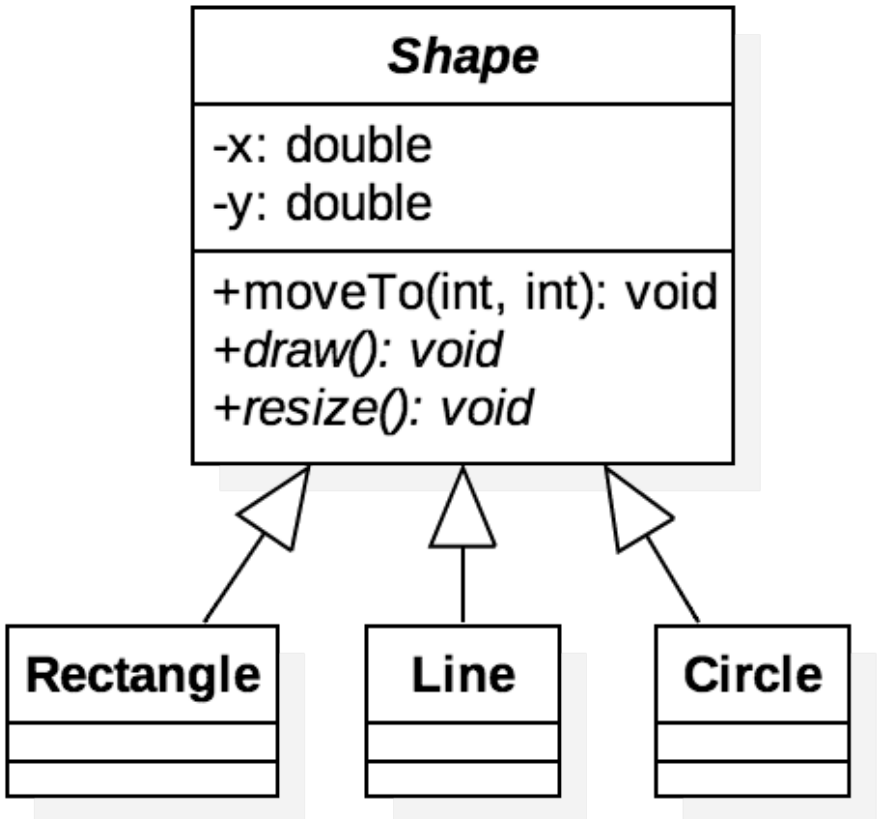
- A final variable is a variable which can be initialized once and cannot be changed later.
  - The compiler makes sure that you can do it only once.
  - A final variable is often declared with static keyword and treated as a constant, for example, **Math.PI**.
- A final method is a method which cannot be overridden by subclasses.
  - You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.
- A class that is declared final cannot be inherited.

Abstract Class

- An abstract class is a class declared abstract.
- The classes that sit at the top of an object hierarchy are typically abstract classes.[15]
- These abstract class may or may not have abstract methods, which are methods declared without implementation.
  - More explicitly, the methods are declared without braces, and followed by a semicolon.
  - If a class has one or more abstract methods, then the class itself must be declared abstract.
- All abstract classes cannot be instantiated.
- Moreover, abstract classes act as placeholders for the subclass objects.

[15] The classes that sit near the bottom of the hierarchy are called concrete classes.

Example



- Abstract methods and classes are in italic.
- In this example, the abstract method *draw*() and *resize*() should be implemented depending on the real shape.

Another IS-A Relationship

- Not all classes share a vertical relationship.
- Instead, some are supposed to perform the specific methods without a vertical relationship.
  - Consider the class **Bird** inherited from **Animal** and **Airplane** inherited from **Transportation**.
  - Both **Bird** and **Airplane** are able to be in the sky.
  - So they should perform the method canFly(), for example.
- By semantics, the method canFly() could not be defined in their superclasses.
- We need a horizontal relationship.

Example

```
1  interface Flyable {
2      void canFly(); // public + abstract
3  }
4
5  abstract class Animal {}
6
7  class Bird extends Animal implements Flyable {
8      public void canFly() {
9          System.out.println("Bird flying...");
10     }
11 }
12
13 abstract class Transportation {}
14
15 class Airplane extends Transportation implements Flyable  {
16     public void canFly() {
17         System.out.println("Airplane flying...");
18     }
19 }
```

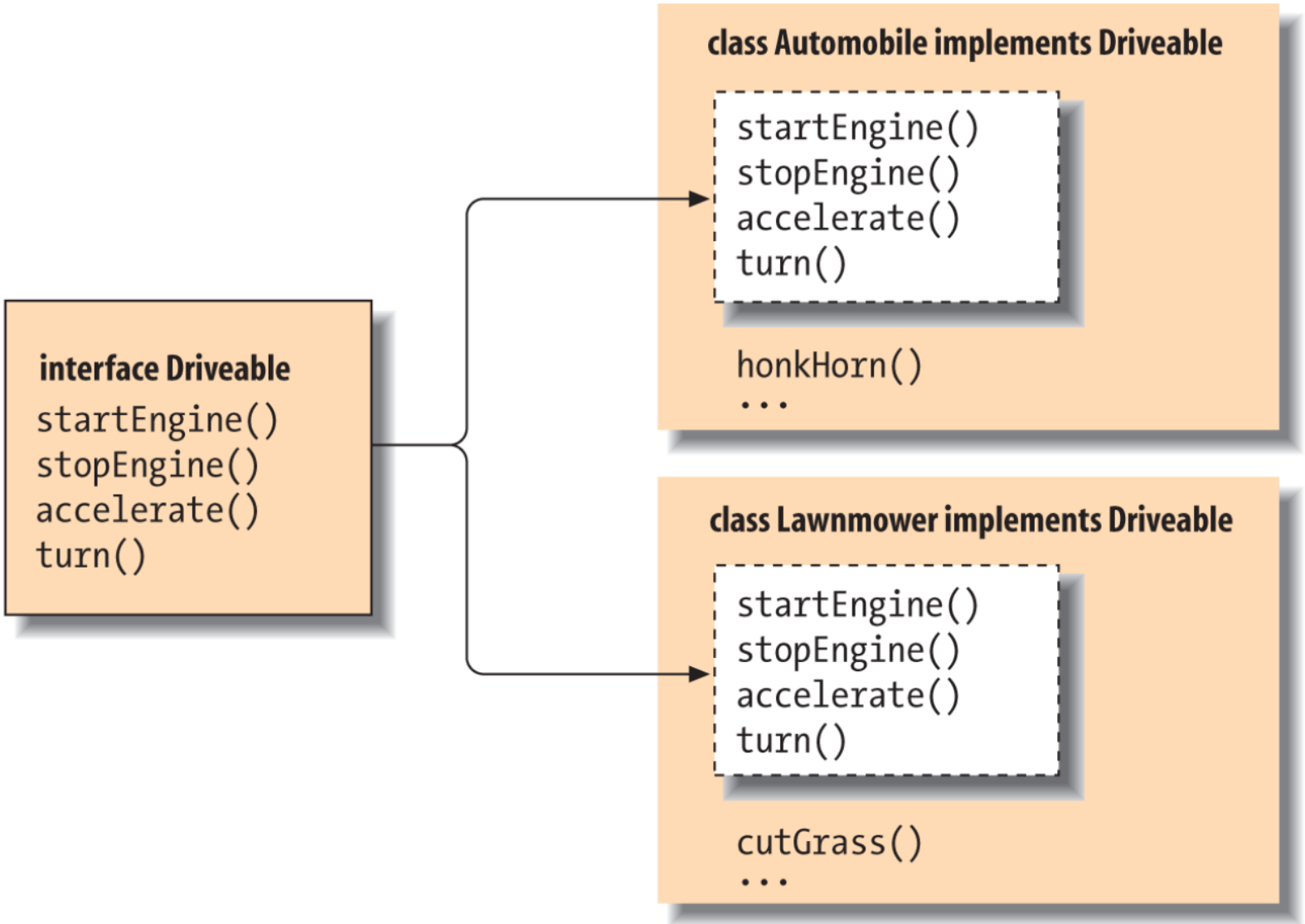
```java
public class InterfaceDemo {
    public static void main(String[] args) {
        Airplane a = new Airplane();
        a.canFly();

        Bird b = new Bird();
        b.canFly();

        Flyable f = a;
        f.canFly(); // output ``Airplane flying...''
        f = b;
        f.canFly(); // output ``Bird flying...''
    }
}
```

Interfaces

- An interface forms a contract between the object and the outside world.
  - For example, the buttons on the television set are the interface between you and the electrical wiring on the other side of its plastic casing.
- An interface is also a reference type, just like classes, in which only method signatures are defined.
- So they can be the types of reference variables!

- Note that interfaces cannot be instantiated (directly).
- A class implementing one or multiple interfaces provides method bodies for each defined method signature.
- This allows a class to play different roles, with each role providing a different set of services.
- For example, combatants in RPG are also the characters who can trade in the market.

Example


interface Driveable
startEngine()
stopEngine()
accelerate()
turn()
class Automobile implements Driveable
startEngine()
stopEngine()
accelerate()
turn()
honkHorn()
...
class Lawnmower implements Driveable
startEngine()
stopEngine()
accelerate()
turn()
cutGrass()
...

Properties of Interfaces

- The methods of an interface are implicitly public.
- In most cases, the class which implements the interface should implement all the methods defined in the interface.
  - Otherwise, the class should be abstract.
- An interface can declare only fields which are static and final.
- You can also define static methods in the interface.
- A new feature since Java SE 8 allows to define the methods with implementation in the interface.
  - A method with implementation in the interface is declared default.

- An interface can extend another interface, just like a class which can extend another class.
  - However, an interface can extend many interfaces as you need.
- For example, **Driveable** and **Updateable** are good interface names.
- Common interfaces are **Runnable**[16], **Serializable**[17], and **Collections**[18].

[16] Related to multithreading.

[17] Aka object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

[18] Collections are related to data structures.

Timing for Interfaces and Abstract Classes

- Consider using abstract classes if you want to:
  - share code among several closely related classes
  - declare non-static or non-final fields
- Consider using interfaces for any of situations as follows:
  - unrelated classes would implement your interface
  - specify the behavior of a particular data type, but not concerned about who implements its behavior
  - take advantage of multiple inheritance

Wrapper Classes

- To treat values as objects, Java supplies standard wrapper classes for each primitive type.
- For example, you can construct a wrapper object from a primitive value or from a string representation of the value.

```
1 ...
2         Double pi = new Double("3.14");
3 ...
```

Primitive	Wrapper
void	java.lang.Void
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

Autoboxing and Unboxing of Primitives

- The Java compiler automatically wraps the primitives in their wrapper types, and unwraps them where appropriate.

```
1 ...
2         Integer i = 1; // autoboxing
3         Integer j = 1;
4         System.out.println(i + j); // unboxing; output 2
5
6         System.out.println(i == j); // output true
7         System.out.println(i.equals(j)); // output true
8 ...
```

- The method **equals**() inherited from **Object** is used to compare the contents of two objects.
  - Herein, the values of wrapper objects.

Immutable Objects

- An object is considered immutable if its state cannot change after it is constructed.
- Often used for value objects.
- Imagine that there is a pool for immutable objects.
- After the value object is first created, this value object is reused if needed.
- This implies that another object is created when we operate on the immutable object.

- For example,

```
1  ...
2          k = new Integer(1);
3          System.out.println(i == k); // output false (why?)
4          System.out.println(k.equals(i)); // output true
5
6          Integer q = 2;
7          i++;
8          System.out.println(i == q); // output true
9          System.out.println(i.equals(q)); // output true
10 ...
```

- Good practice when it comes to concurrent programming.[19]
- Another example is String objects.

[19] See http://www.javapractices.com/topic/TopicAction.do?Id=29.

enum Types[20]

- An enum type is an reference type limited to an explicit set of values.
- An order among these values is defined by their order of declaration.
- There exists a correspondence with string names identical to the name declared.

[20] The keyword enum is a shorthand for enumeration.

Example

```
1 ...
2 enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday,
      Friday, Saturday}
3 ...
```

- Actually, **Weekday** is a subclass of enum type with seven static and final objects corresponding to the seven enumerated values.
- The **Weekday** instances which really exist are the seven enumerated values.
- So this mechanism enhances type safety!

```java
public class EnumerationDemo {
    public static void main(String[] args) {
        Weekday[] weekdays = Weekday.values();
        // The method values() returns a Weekday array.

        for (Weekday day: weekdays) {
            System.out.println(day);
        }

        Weekday today = Weekday.Sunday;
        Weekday tomorrow = Weekday.Monday;

        System.out.println(today == tomorrow); // output false
    }
}
```

Exercise: Colors

```
1  enum Color {
2
3      Red, Green, Blue; // three options
4
5      static Color randomColor() {
6          Color[] colorSet = values();
7          int pickOneColor = (int) (Math.random() * colorSet.
               length);
8          return colorSet[pickOneColor];
9      }
10 }
11
12 public class EnumDemo {
13     public static void main(String[] args) {
14         for(int i = 1 ; i <= 3; i++)
15             System.out.println(Color.randomColor());
16     }
17 }
```

Exercise: Size

```
1  enum Size {
2
3      Large("L"), Medium("M"), Small("S");
4
5      private String abbr;
6      private Size(String abbr) { this.abbr = abbr; }
7
8      public String getAbbr() {
9          return this.abbr;
10     }
11 }
12
13 public class EnumDemo {
14     public static void main(String[] args) {
15         System.out.println(Size.Small.getAbbr()); // output S
16     }
17 }
```

Packages

- We organize related types into packages for the following purposes:
  - To make types easier to find and use
  - To avoid naming conflicts
  - To control access
- For example, fundamental classes are in **java.lang** and classes for I/O are in **java.io**.

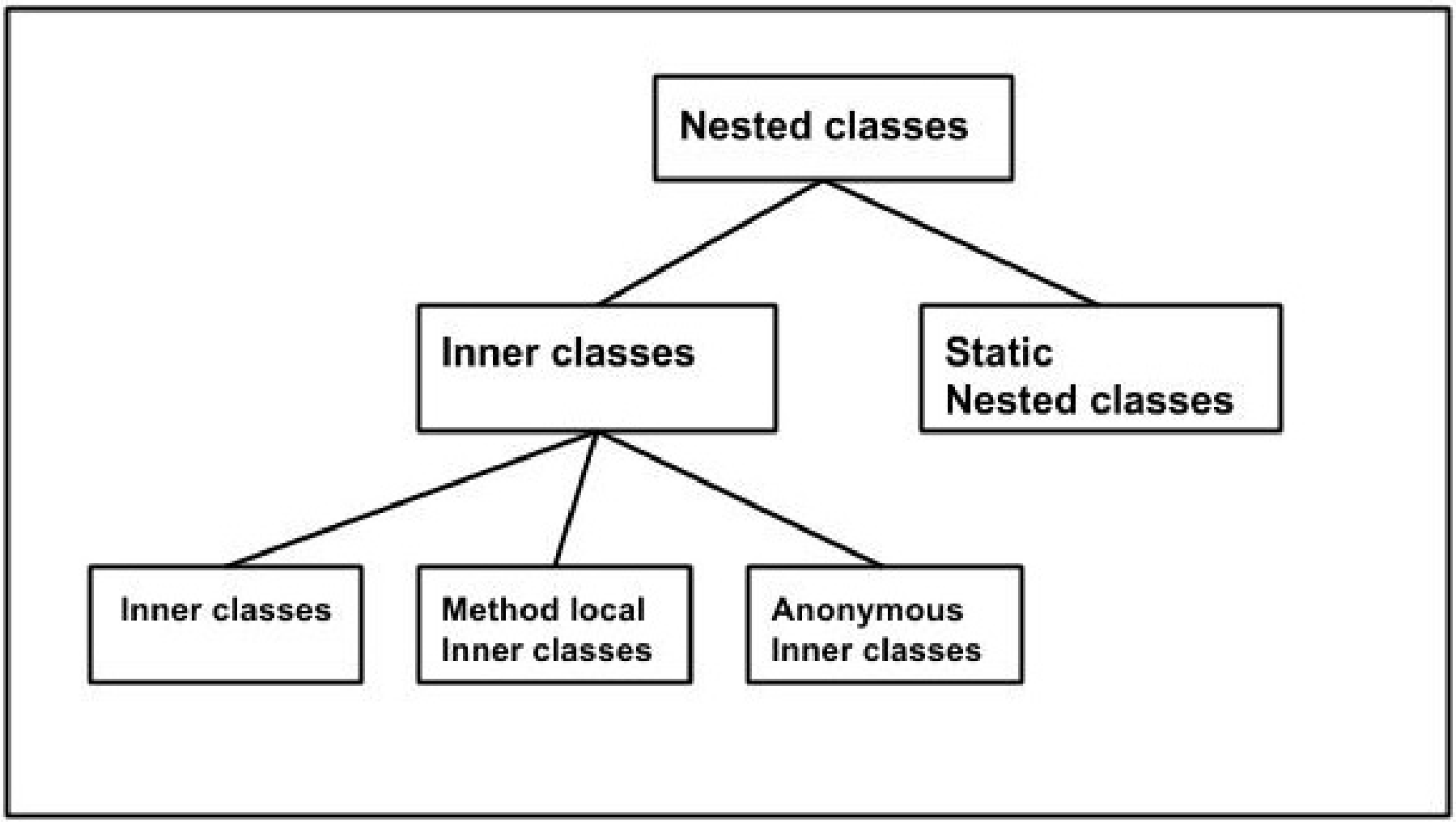
Access Control

Scope \ Modifier	private	(package)	protected	public
Within the class	✓	✓	✓	✓
Within the package	x	✓	✓	✓
Inherited classes	x	x	✓	✓
Out of package	x	x	x	✓

Nested Classes

- A nested class is a member of its enclosing class.
- Non-static nested classes, aka inner classes, have access to other members of the enclosing class, even if they are declared private.
- Instead, static nested classes do not have access to other instance members of the enclosing class.
- Timing of usage:
  - Logically grouping classes that are only used in one place
  - Increasing encapsulation
  - Leading to more readable and maintainable code

Family of Nested Classes



Inner Classes

- Inner classes can be classified depending on how and where you define them:
  - Inner class
  - Method-local inner class
  - Anonymous inner class
- Unlike a normal class[21], an inner class can be declared private.
- Note that the creation of inner-type objects is available after the outer-type object is created.
  - In other words, you cannot invoke the constructor of the inner type without having the outer type object.
- For static members in the inner classes,
  - you can declare a static member which is supposed to be final;
  - however, static methods can only be declared in a static or top level type.

[21] We call these the top classes.

Example: Inner Class

```
1  class OuterClass {
2      private int x = 1;
3      InnerClass y = new InnerClass();
4
5      class InnerClass {
6          public void print() {
7              System.out.println(x); // ok!
8          }
9      }
10 }
11
12 public class InnerClassDemo {
13     public static void main(String[] args) {
14         OuterClass outer = new OuterClass();
15         outer.x.print(); // output 1
16
17         InnerClass inner = new InnerClass(); // oops
18         // Since InnerClass type cannot be resolved out of
               OuterClass.
19         outer.new InnerClass().print(); // output 1
20     }
21 }
```

Example: Method-local Inner Class

```
1  class OuterClass {
2      private int x = 1;
3
4      void outerClassMethod() {
5          class MLInnerClass { // should be in the beginning
6              int y = 2;
7              static int z = 3; // implicitly final
8
9              void print() {
10                 System.out.println(x);
11                 System.out.println(y);
12                 System.out.println(z);
13             }
14         }
15
16         MLInnerClass w = new MLInnerClass();
17         w.print();
18     }
19 }
```

Anonymous Inner Class

- Anonymous inner classes are an extension of the syntax of the new operation, enabling you to declare and instantiate a class at the same time.
  - However, these do not have a name.
- Use them when you need to use these types only once.

Example

```
1  abstract class A {
2      void foo();
3  }
4
5  public class AnonymousClassDemoOne {
6      public static void main(String[] args) {
7          A a = new A() {
8              public void foo() { /* different implementation */ }
9              void helper() { /* a subroutine for foo */ }
10         };
11
12         a.foo();
13     }
14 }
```

- You may invoke a.foo() but not a.helper() because helper() is not defined in class A.

Exercise

```
1  interface B {
2      void foo();
3  }
4
5  public class AnonymousClassDemoTwo {
6      public static void main(String[] args) {
7          B b = new B() {
8              public void foo() { /* different implementation */ }
9          };
10
11         b.foo();
12     }
13 }
```

- An interface can be used to instantiate an object indirectly by anonymous classes with implementing the abstract methods.

Iterators

- An important use of inner classes is to define an adapter class as a helper object.
- Using adapter classes, we can write classes more naturally, without having to anticipate every conceivable user's needs in advance.
- Instead, you provide adapter classes that marry your class to a particular interface.
- For example, an iterator is a simple and standard interface to enumerate objects in many data structures.
    - The **java.util.Iterator** interface defines two methods: public boolean hasNext() and public **Object** next().

Example: An Iterator

```
1  class Box implements Iterable {
2
3      int[] arr = {1, 2, 3};
4      Iterator iter = new Iterator() {
5          int count = 0;
6
7          public boolean hasNext() {
8              if (count < arr.length)
9                  return true;
10             else
11                 return false;
12         }
13
14         public Object next() {
15             return arr[count++];
16         }
17     };
18
19     public Iterator iterator() {
20         return iter;
21     }
22 }
```

```java
import java.util.Iterator;
import java.util.Scanner;

public class IteratorDemo {
    public static void main(String[] args) {
        Box b = new Box();
        for (Object x: b) {
            System.out.println(x);
        }
    }
}
```

Static Nested Class

- A static inner class is a nested class which is a static member of the outer class.
  - So they can access to other static members without instantiating the outer class.
- Just like static members, a static nested class does not have access to the instance members of the outer class.
- Most important, a static nested class can be instantiated directly, without instantiating the outer class object first.
  - Static nested classes act something like a minipackage.

Example

```
1  class OuterClass {
2      static int x = 1;
3      int y = 2;
4
5      void OuterClassMethod() {
6          System.out.println(y);
7      }
8
9      static class StaticNestedClass {
10         int z = 3;
11         void StaticNestedClassMethod() {
12             System.out.println(x);
13             System.out.println(y); // Oops, static members
                   cannot access to instance members.
14             System.out.println(z);
15         }
16     }
17 }
```

```
1 public class StaticNestedClassDemo {
2     public static void main(String[] args) {
3         OuterClass.StaticNestedClass x = new OuterClass.
              StaticNestedClass();
4         x.StaticNestedClassMethod();
5     }
6 }
```

Classpath[22]

- The variable **classpath** is an environment variable for the Java compiler to specify the location of user-defined classes and packages.
  - By default, only the packages of the JDK standard API and extension packages are accessible without needing to set where to find them.
- The path for all user-defined packages and libraries must be set in the command-line (or in the Manifest associated with the JAR file containing the classes).

[22] `https://en.wikipedia.org/wiki/Classpath_(Java)`

Usage of Classpath

- You may use the following command in any terminal:

  java -cp [the absolute path of the classes or packages] [the full name of the application to run]

- For Windows users, try

  java -cp c:\workspace\project train.java.HelloWorld

- On Linux/Unix/Mac OS users, try

  java -cp /workspace/project train.java.HelloWorld

Java Archive (JAR)[24]

- JAR is a packed format typically used to aggregate many Java class files, associated metadata[23] and resources (text, images, etc.) into one file to distribute the application software or libraries running on the Java platform.
  - Try an executable JAR!

[23] Metadata refers data of data.

[24] See `https://docs.oracle.com/javase/tutorial/deployment/jar/`.