Java Basics

These slides contain a fast summary of Java syntax for people who already know some programming.

Where's the Source Code?

In Java, all source code is contained in classes.

A class defines a *kind of object*.

and the object's attributes and behavior.

You create objects (instances) from a class.

Creating Objects

Use "new" to create an instance (object) of a class.

```
new Date( )
```

To refer to the object again later, you usually want to assign a *reference* to it:

```
Date d = new Date( );
```

What does "new Date( )" mean? How about this:

```
Date d = new Date(112, 2, 20);
```

Answer: it depends on the source code.

Defining your own class

To define a new <u>kind</u> of object, you create a Java *class.*

Example:

in the coin purse project, we want to have "coins" that remember their value, so we define a Coin class.

Class Structure

```
package coinpurse;
/**
 * Describe this class.
 * @author Your Name
 */
public class Coin {
    static attributes

    instance attributes

    constructors

    methods
}
```

Attributes

Attributes are what an object knows.

To refer to something, it must be a variable.

```
package coinpurse;
public class Coin {
   private double value;
   private String currency;



}
```

attributes of a Coin:

a Coin has a value and currency.

Declaring Attributes

```
public class Coin {
   /** value of coin */
   private double value;
```

Javadoc for attribute

Visibility

- public
- protected
- (default)
- **private**

Data Type

- primitive
- class name
- interface
- array

Variable Name

- name of attribute
- should start with lowercase

Common Java Data Types

Some data types used in Java are:

Data Type	Examples
int	-100 ... -1 0 1 2 ... 2147483647
double	0.5 -3.70 2.98E+8
boolean	true false
String	"Hello" "I'm hungry" "turn left"

List
ArrayList
	Collection of things.
	
	List list = new ArrayList( );
	
	list.add("apple"); list.add("orange");

Initialize All Your Attributes!

```
public class Coin {
   private double value;
   private String currency = "THB";

   /** initialize a new coin */
   public Coin( double value ) {
      this.value = value ;
   }
```

Two ways to initialize attributes:

- assign a value as part of declaration, or
- (better) initialize in a constructor

Constructor Initializes a New Object

```
Coin ten = new Coin( 10 );
```

```
/** initialize a new coin */
public Coin( double value ) {
    this.value = value ;
}
```

Constructor has the same name as the class.

Constructor does not have any return value. Not even "void".

"this" means "this object". "this" is used to *distinguish* between the parameter value and attribute value.

How Objects are Created

```
new Coin( 10 )
```

Java creates object in memory

JVM invokes a *constructor* initialize state of the object

```
// constructor's job is to
// initialize a new object
public Coin( double val ) {
  this.value = val;
}
```

Correct this Code

```
public class Coin {
   private double value;
   public void Coin(double val) {
        this.value = val;
   }
```

This code has legal syntax, but it is not a constructor.

More than One Constructor

```
public class Coin {
  /** default constructor */
  public Coin( ) {
    this.value = 0;
    this.currency = "THB";
  }
  public Coin(double value) {
    this.value = value;
    this.currency = "THB";
  }
  public Coin(double value,
          String currency) {
    ...
```

A class can have *many constructors*,

if they have different parameters.

Default Constructor

```
public class Coin {
  private double value;
  private String currency;
  public Coin( ) {
     this.value = 0 ;
     this.currency = "THB";
  }
```

```
Coin zero = new Coin(  );
```

A constructor with no parameters is called the default constructor.

Avoid Duplicate Code

```
public class Coin {
  /** default constructor */
  public Coin( ) {
    this.value = 0;
    this.currency = "THB";
  }
  public Coin(double value) {
    this.value = value;
    this.currency = "THB";
  }
  public Coin(double value,String currency){
    this.value = value;
    this.currency = currency;
```

These 3 constructors all do the same thing.

Constructor calls Constructor

A constructor can call another constructor using "this()", but it must be the first statement in constructor.

```
public Coin( ) {
   this( 0, "THB");
}
public Coin(double value) {
   this( value, "THB");
}
public Coin(double value, String curr) {
   if (value < 0)
      throw new IllegalArgumentException(...);
   this.value = value;
   this.currency = curr;
}
```

Methods

- ✓ The behavior of objects is defined in methods.
- ✓ Methods contain the program's logic.

name of method

```
public String toString(  ) {
    return String.format("%d %s coin",
        this.value, this.currency );
    //ex: 5 Baht coin
  }
```

instructions for this method

Method in Java

return value (nothing)

name of the method

start of this method

```
   public void act( ) {
         .
         .
         .
         .
   }
```

instructions

of the method ("body")

end of this method

The Body of a Method

The body of a method is a **list of instructions**.

Instructions are executed from top to bottom.

```
public void act( ) {
  move( );
  turn( 30 );
  move( );
}
```

list of instructions

";" ends each instruction

You can use a { block } anywhere

You can use { } for "else" or "while" or ...

```
if ( balance > 0 ) {
        block of statements for
        "then" case

}
else {
        block of statements for
        "else" case
}
```

else block

Writing a Method that Returns Result

this method returns an "int" value

```
public class Coin {
  private int value;
  /** compare 2 coins by value */
  public int compareTo(Coin other) {
    int diff = this.value - other.value;
    return diff;
  }
}
```

Method with a Parameter

We use *parameters* to give information to a method.

Behavior in English with *parameter*	Method in Java with *parameter*
*turn **left***	`turn( -90 )`
*turn **15 degrees***	`turn( 15 )`
*can see **a Worm** ?*	`canSee( Worm.class )`
*move to **x , y***	`setLocation( x, y )`

Writing a Method with Parameter

specify the *data type*
of the parameter value

the parameter name

```
/* add value of two coins */
int add( Coin coin1, Coint coin2 ) {
   int sum = coin1.value + coin2.value;
   return sum;
}
```

Attributes for Knowing Things

An object has to remember information.

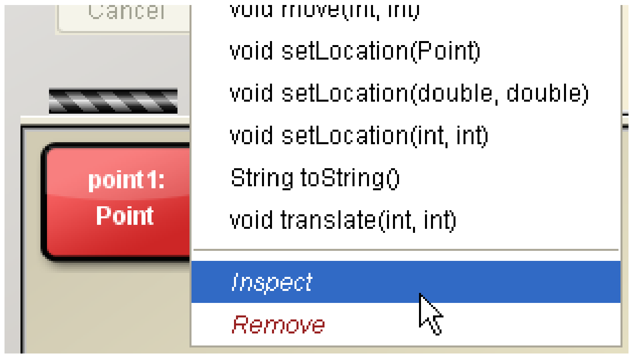
A class defines the attributes of a kind of object.

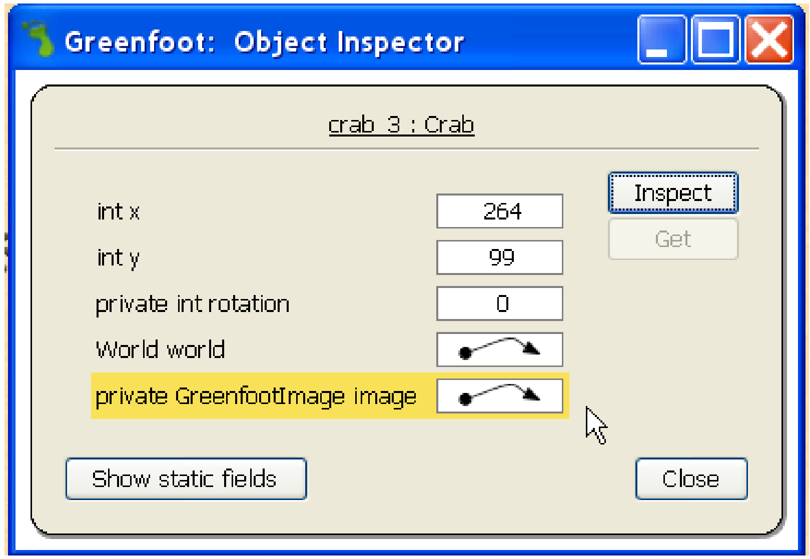
See *attributes* of an Object

In BlueJ, you can "inspect" attributes of an object.

1. Create an object, e.g. java.awt.Point
2. Right click and choose "*Inspect*".
3. What are the attributes?

The attributes of an object are also called "fields" or "properties".





int x	264
int y	99
private int rotation	0
World world	↗
private GreenfootImage image	↗

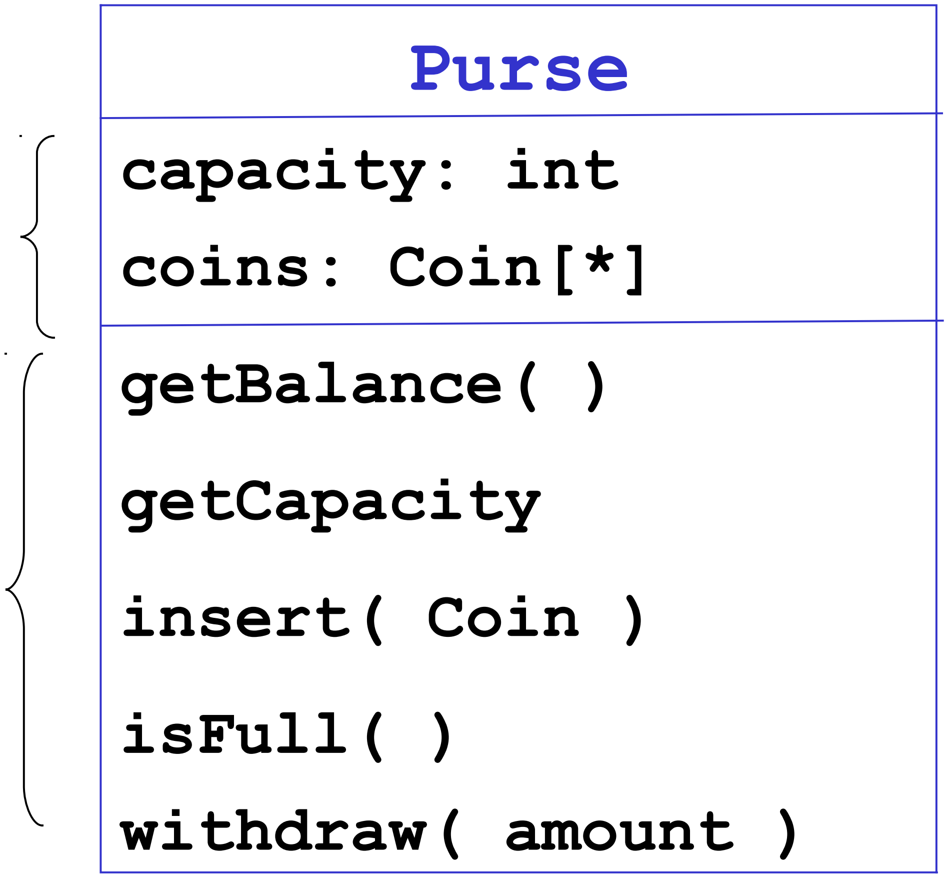
Attributes are what an object knows

Attributes -

what a Purse knows

Methods -

what a Purse can do

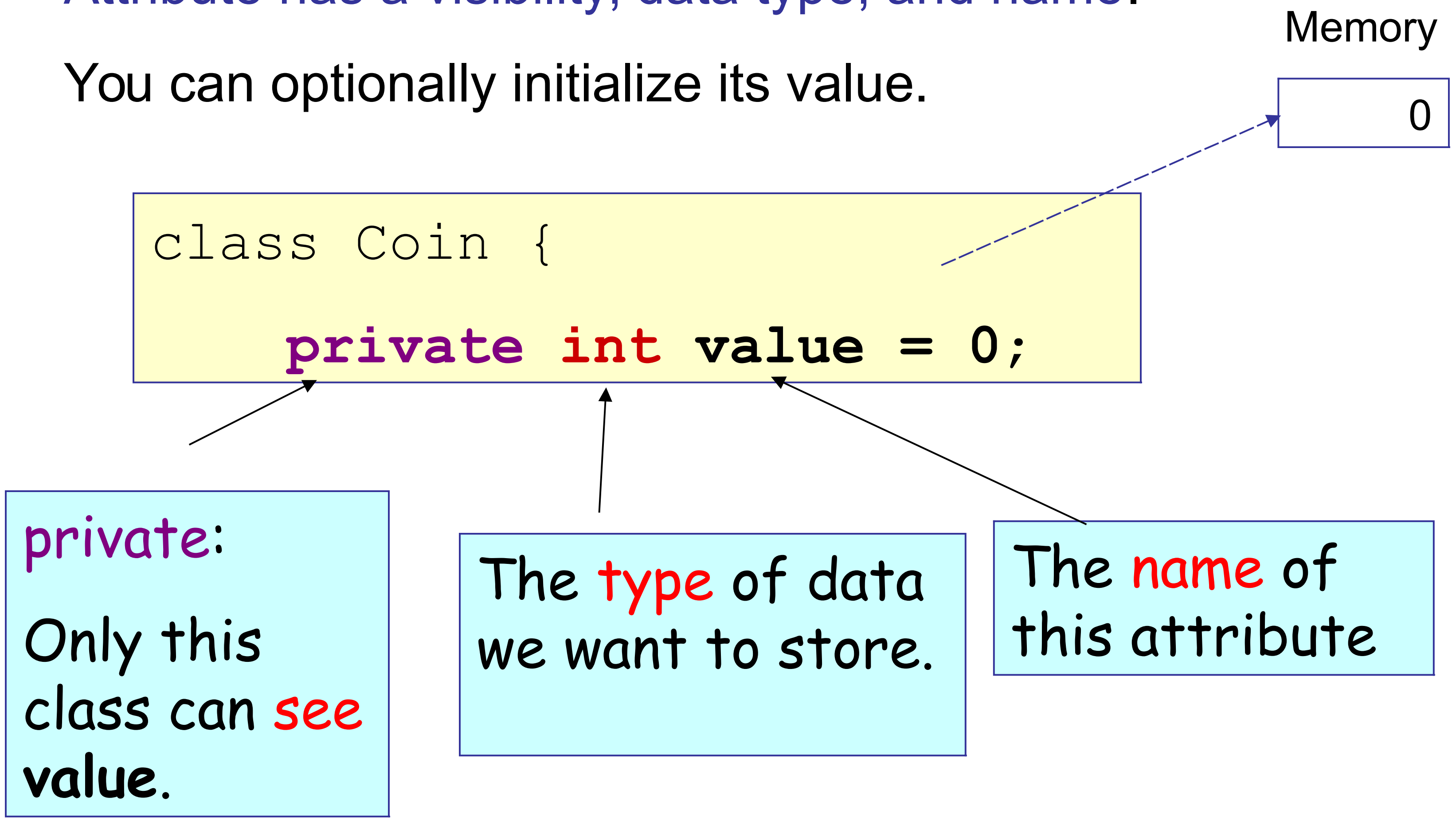


Defining an Attribute

Attributes should be defined near top of class.

Attribute has a visibility, data type, and name.

You can optionally initialize its value.



```
class Coin {

    private int value = 0;
```

private:

Only this class can see **value**.

The type of data we want to store.

The name of this attribute

Assigning and Changing a Value

We can change the value of a variable as often as we like. To assign a value use:

```
variableName = some expression;
```

variable = *assign to* *expression*

Memory

```
count = 0;

count = count + 1;
```

0

1

Values and References

- An attribute (variable) of a primitive type like "int" contains a value of the primitive.
- An attribute (variable) of an object type like Coin is a reference.

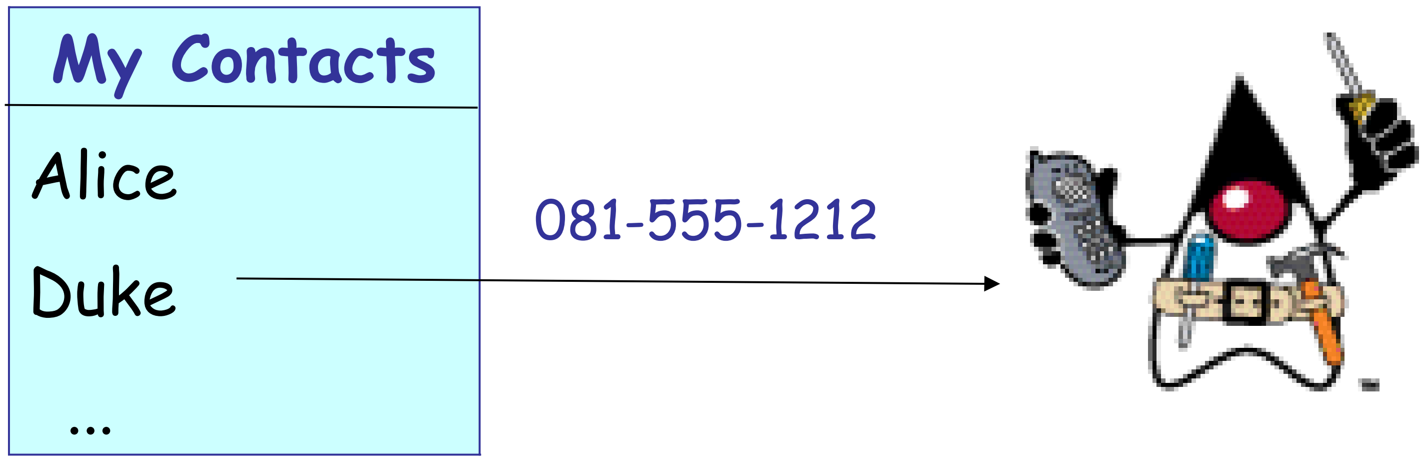
Variables as *References*

A variable can be used to *remember* another object.

- A reference (variable) is how one object sends a message to another object.

Example:

A mobile phone contact is a *reference* to another mobile phone ...



Variables as References (2)

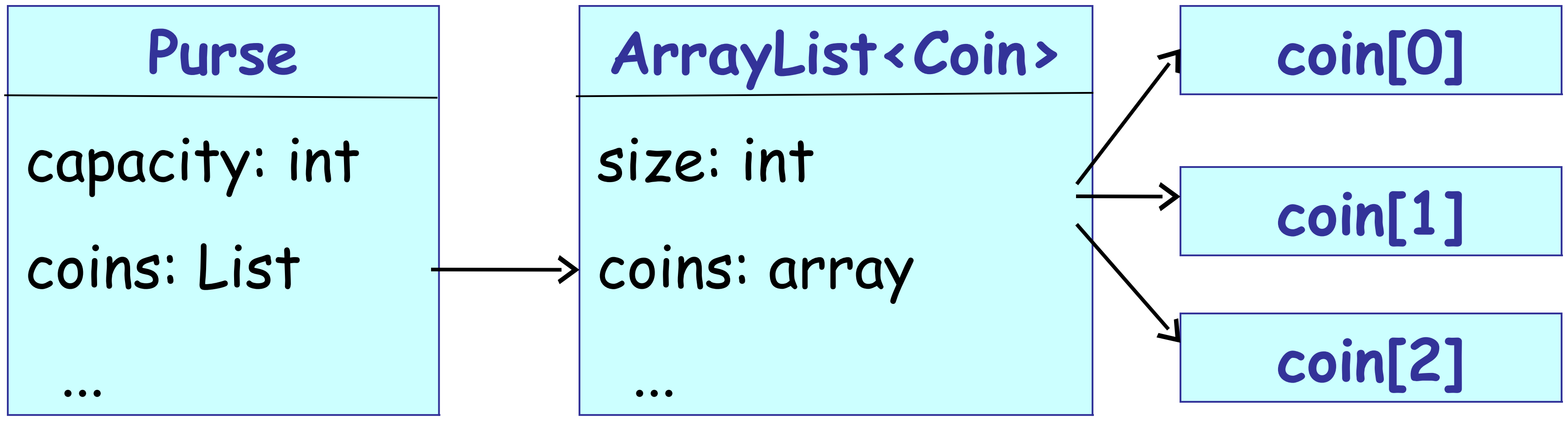
A variable is a reference to another object.

Example:

A Purse contains a *reference* to a List of coins.

The List contains *references* to Coin objects.

A Purse has a capacity which is just a value (int).



Variables as References (3)

Use a reference to ask as object some questions, using the object's methods.

```
void describe(Purse purse) {
   int balance = purse.getBalance();
   if ( purse.isFull() ) ...
```

Local Variables

Variables defined inside a method are local variables.

(1) can only be used *inside the method*

(2) deleted when the method returns

```
public class Purse {

  public int getBalance( ) {
    int balance = 0;
    for(int k=0; k<coins.size(); k++) {
        // add coins.get(k) to balance
    }
```

Local variables are defined inside a method.

3 Types of Variables

An object has access to 3 kinds of variables:

Attributes of the object

Static attributes of the class

Local variables and parameters (inside one method)

Local Variables vs. Attributes

An **attribute** is something an object *remembers* for its whole life.

A **local variable** is for temporary data. The value is lost when execution leaves the method.

```
public class Purse {
   private int capacity;
   private List coins;
   public int getBalance( ) {
      int balance = ...;
      return balance;
   }
```

A purse must *remember* its capacity and coins

balance can be computed each time we ask for it.
Don't need to remember.

Static Method as Service

Some classes provide a "service".

A service is something that the class does, but is not associated with any object.

Services are defined by *static methods*.

Get the current system time in milliseconds:

```
System.getTimeMillis(  );
```

Name of Class

static method name

Service: method without an object

Some other service (static) methods:

Square root:

```
double r = Math.sqrt( 2 );
```

Convert a String to an integer:

```
int value = Integer.parseInt("123");
```

Play a sound in Greenfoot:

```
Greenfoot.playSound("starwars.wav");
```

These methods are performed by a class, not an object:

Service methods are static

A method that doesn't belong to an object is called static.

`Math.sqrt( )` - static method in the Math class

`Greenfoot.playSound( )` static method **in** Greenfoot

To create a static method, add the word "**`static`**":

```
/** distance between points (x1,y1) and (x2,y2) */
public static double distance( x1, y1, x2, y2 ) {
   // hypot computes hypothenous of a triangle
   double d = Math.hypot( x1 - x2, y1 - y2 );
   return d;
}
```

Java Naming Convention

class name begins with Uppercase: `Coffee, String`

method name uses camelCase: `getMoreCoffee( )`

variable name also uses camelCase: `myCoffee`

constants use UPPER_CASE and _: `MAX_COFFEE`

package names are all lowercase (but not always):

```
java.util  java.io
org.apache.commons.logging
```

primitive type names are all lowercase:

```
boolean, int, double
```

What are these?

Date

System

System.nanoTime( )

System.out

System.out.println( )

double

Double

"Hello nerd".length( )

java.lang.Double.MAX_VALUE

Comparable

java.util

java.util.ArrayList

java.util.*List*

Is it a ...

package

class

primitive type

attribute ("field")

method

(static or instance)

constant

(static final attribute)

interface *(more advanced)*

???

Packages

- Java uses packages to organize classes.
- Packages reduce size of *name space* and avoid *name conflicts (two classes with same name)*

Example: there are 2 `Date` classes.

```
java.util.Date  "Date" class in java.util
java.sql.Date   "Date" class in java.sql
```

To use the Date from java.utll package, write:

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

Core Packages

`java.lang`	Java language core classes. `Object, String, System, Integer, Double, Math, Thread` java compiler always imports this package, so you don't need to.

`java.io`
`(and`
`java.nio)`
	Classes for input and output
	`InputStream, BufferedReader, File, OutputStream`

`java.util`	Date/time classes, collections, & utilities
	`Calendar, Date,`
	`List, ArrayList, Set,`
	`Arrays, Formatter, Scanner`

Importing classes

Write "import" statements at top of file, **after** the "package" statement (if you have one).

```
package coinpurse;
import java.util.Scanner;
import java.util.Date;
/**
 * User interface for coin purse.
 */
public class ConsoleDialog {
   Scanner console = new Scanner( System.in );
   ...
```

imports come after package statement and before class Javadoc comment.

Importing all classes

Write "import" statements at top of file, **after** the "package" statement (if you have one).

```
package coinpurse;
import java.util.*;
/**
 * User interface for coin purse.
 */
public class ConsoleDialog {
  Scanner console = new Scanner( System.in );
  ...
```

imports come after package statement and before class Javadoc comment.

What is "import"?

`import` tells the compiler *where* to find classes.

It doesn't actually "import" any code!

```
package guessinggame;
import java.util.Random;
/**
 * User interface for guessing game.
 */
public class GameDialog {
   private Random rand = new Random( );
   ...
```

tell the compiler where to find the Random class

Why `import`?

```
import java.util.Date;
class Appointment {
   private Date startDate;
```

The reason for "import" to to resolve ambiguity.

Many classes can have the *same name*.

Java API has 2 classes named "`Date`".

`java.util.Date` and `java.sql.Date`.

3 classes named "`Timer`"

5 "`Element`" classes and interfaces.

Import Everything

You can import everything from a package. Use *

```
package lazyimport;
import java.util.*;
import java.io.InputStream;

class Person {
   private static Scanner console = ...;
   private Date birthday;
   private List<Person> friends;
   ...
```

Ambiguity in Import

If a class matches more than one wildcard "*", Java requires you to resolve the ambiguity using an import without the wildcard.

Example: There are 2 Date classes: `java.util.Date` and `java.sql.Date`. These imports are *ambiguous:*

```
import java.util.*;
import java.sql.*;
/** a class using a Date */
class Ambiguous {
  private Date today;
```

which **Date** class should Java use?

Resolving Ambiguity

There are two ways to resolve ambiguity.

1. import a specific class (no wildcard)
2. use the fully qualified name in Java code

```
import java.util.*;
import java.sql.*;
import java.util.Date; // Solution #1
class Ambiguous {
  private Date today = new Date( );
      // Solution #2
  private java.sql.Date mdate
      = new java.sql.Date( );
```

Array versus ArrayList (a List)

Array

```
// array of coins
Coin[] coins;
coins = new Coin[10];
coins[0] = new Coin(5);
coins[1] = new Coin(20);
System.out.println( coins[4] );  // print null
```

ArrayList is a kind of List

A List can hold any amount of data.

ArrayList is a kind of list.

List and ArrayList are in java.util.

```
// array of coins
Coin[] coins;
coins = new Coin[10];
coins[0] = new Coin(5);
coins[1] = new Coin(20);
System.out.println( coins[4] );  // print null
```

Console Input and Output

Display output

```java
System.out.println("I'm a string" + " again");

System.out.print("apple");
System.out.print("banana\n");
System.out.print("grape");
```

```
I'm a string again
applebanana
grape
```

Input

System.in can only read bytes. Not very useful.

```
int c = System.in.read( ); // read 1 byte
byte[] b;
System.in.read( b );  // array of bytes
```

Use a Scanner to read input as int, double, String, etc.

```
Scanner console = new Scanner(System.in);
String word = console.next();
String line = console.nextLine();
int number = console.nextInt();
double x = console.nextDouble();
```

Packaging and Commenting Code

```
package coinpurse;
/**
 * Coin represents money with an integer value.
 * @author Bill Gates
 */
public class Coin {
  private int value;
  /**
   * Initialize a new coin object.
   * @param value is the value of the coin
   */
  public Coin( int value ) {
    this.value = value;
  }
```

Complex logic: and or not

The test expression of "if" may contain `&&` (and), `||` (or), and `!` (not), as long as the result is true or false.

```
int x = getX( );
int y = getY( );
// if x ≤ 0 or y ≤ 0 then turn right
if ( x <= 0 || y <= 0 )
    turn( +15 );
// if we are hungry and see a worm...
if ( hungry( ) && canSee(Worm.class) )
    eat(Worm.class);
```

Summary (1)

- ✓ A compiler translates Java source code into a form that can be run.
- ✓ An object-oriented program consists of classes.
- ✓ Classes can contain:

  attributes of objects -- things an object knows

  methods -- behavior of objects

  constructor -- initializes data of a new object

  static methods -- services provided by the class

  static variables -- things known by the class

Summary (2)

- A class defines a kind of object, like Actor or Crab.
- The methods of a class contain the logic for how an object behaves (written in Java).
- A method can call other methods in the same object, e.g. `act( )` calls `move( )`.
- A method can call methods of other objects, e.g. `atWorldEdge()` calls `world.getWidth( )`.

Question: why { ... } ?

Why do we have to write { and } around the method instructions?

Why?

```
public void sayHello(String who) {
  System.out.println( "Hello "+who );
}
```

Why?

How to convert number to String?

How to convert a number n to a String?

```
int n = 100;
String s = n;   // error: must convert to string


// At least 4 possible solutions:
String s1 =
String s2 =
String s3 =
String s4 =
```

How to convert a number to String?

How to convert a number n to a String?

```
int n = 100;
String s = n;   // ERROR: must convert to string


// At least 4 solutions:
String s1 = Integer.toString( n );
String s2 = "" + n;
String s3 = String.valueOf( n );
String s4 = String.format( "%d", n );
```

Summary about Methods

If you already understand how to use methods and parameters, you can skip this part.

Anatomy of a method (1)

Who can use this method?

public = any one

protected = me and my chidren (subclasses)

private = only my class

```
public void sayHello( ) {
   String who = "Cat";
   System.out.println( "Hello "+who );
}
```

Anatomy of a method (2)

What answer (value) is returned?

void = nothing returned

int = returns an integer (0, 1, 2, ...)

etc. a method can return *anything*

```
public void sayHello( ) {
   String who = "Cat";
   System.out.println( "Hello "+who );
}
```

Anatomy of a method (3)

Name of this method

```
public void sayHello( ) {
   String who = "Cat";
   System.out.println( "Hello "+who );
}
```

Anatomy of a method (4)

Parameters for sending info to this method (none here).

```
public void sayHello( ) {
   String who = "Cat";
   System.out.println( "Hello "+who );
}
```

Anatomy of a method (5)

Parameter
  who to greet?
Method wants a String.

```
public void sayHello(String who) {
  System.out.println( "Hello "+who );
}
```

```
sayHello( "Cat" );
sayHello( "Bird" );
sayHello( 2.5 );  ERROR
```

Anatomy of a method (6)

Return a boolean (true or false)

Parameter (info): what kind of food to look for?

```
public boolean canSee(Class food) {
    Object obj =
            getOneIntersectingObject(food);
    if (obj != null) return true;
    else return false;
}
```

Call another method, from the Actor class.