Inheritance

CS102A Lecture 10

James YU
yujq3@sustech.edu.cn

Department of Computer Science and Engineering
Southern University of Science and Technology

Nov. 16, 2020

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

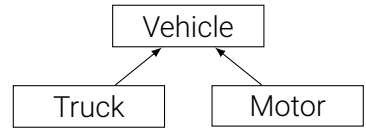
Objectives

- Inheritance
- Superclass and subclass

What exactly is inheritance?

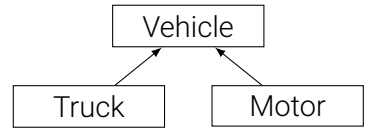
```java
public class Vehicle {
  public int number_wheels = 4;
  public void does() {
    System.out.println("Transporting");
  }
}
```

```java
public class Truck extends Vehicle {
  public int size_container= 500;
}
```



What exactly is inheritance?

```java
public class TrunkTest {
  public static void main(String args[]) {
    Truck obj = new Truck();
    System.out.println(obj.size_container);
    System.out.println(obj.number_wheels );
    obj.does();
  }
}
```



Inheritance

- Consider a scenario where you have carefully designed and implemented a `Vehicle` class, and you need a `Truck` class in your system. Will you create the new class from scratch?

On one hand, trucks have some traits in common with many vehicles. Some code can be shared. (So no?)

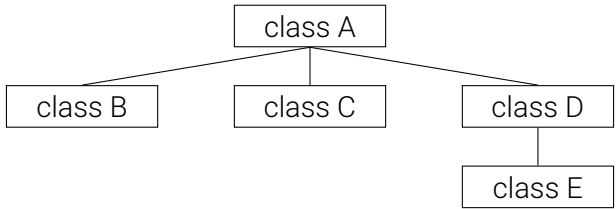
On another hand, trucks have their own characteristics e.g., two seats, can carry huge things. (So yes?)

Inheritance

- A form of software reuse in which a new class is created by absorbing an existing class' members and embellishing them with new or modified capabilities.
- Can save time during program development by basing new classes on existing high-quality software.
- Increases the likelihood that a system can be implemented and maintained effectively.

Inheritance

- When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class.
  - Existing class is the superclass.
  - New class is the subclass
- Each subclass can be a superclass of future subclasses, forming a class hierarchy.

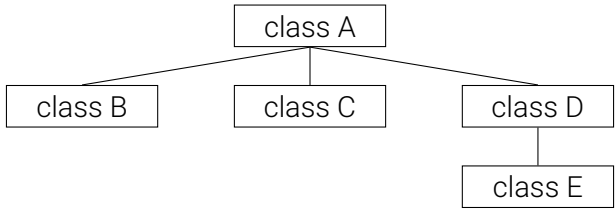


Inheritance

- A subclass can add its own fields and methods.
- A subclass is *more specific* than its superclass and represents a more specialized group of objects.
- The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.
  - This is why inheritance is sometimes referred to as *specialization*.

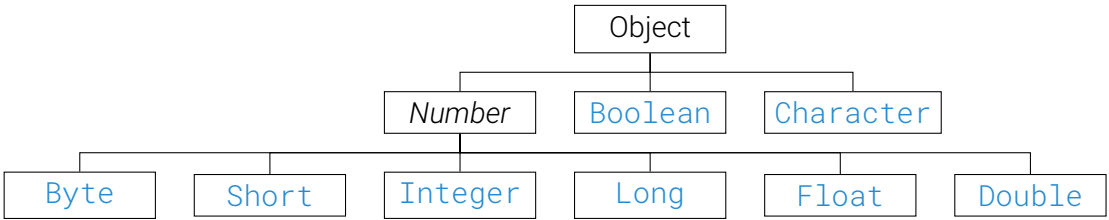
Inheritance

- The direct superclass is the superclass from which the subclass explicitly inherits (A is the direct superclass of C).
- An indirect superclass is any class above the direct superclass in the class hierarchy (e.g., A is an indirect superclass of E).



Inheritance

- The Java class hierarchy begins with class `java.lang.Object`.
  - Every class directly or indirectly extends (or "inherits from") `Object`.
- Java supports only single inheritance, in which each class is derived from exactly one direct superclass.



Inheritance vs. Composition

- Inheritance: *Is-a* relationship between classes.
  - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass (a truck is also a vehicle)
- Composition: *Has-a* relationship between classes.
  - In a *has-a* relationship, an object contains as members references to other objects (a house contains a kitchen)

Superclass and Subclass

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

- Objects of all classes that extend a common superclass can be treated as objects of that superclass (e.g., `java.lang.Object`).
  - Commonality expressed in the members of the superclass.
- Inheritance issue
  - A subclass can inherit methods that it does not need or should not have.
  - Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method.
  - The subclass can override (redefine) the superclass method with an appropriate implementation.

`public` and `private` members

- A class' `public` members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- A class' `private` members are accessible only within the class itself (invisible to subclasses).

`protected` members

- `protected` access is an intermediate access level between `public` and `private`.
- A superclass' `protected` members can be accessed by
  - members of that superclass,
  - members of its subclasses,
  - members of other classes in the same package.
- All `public` and `protected` superclass members retain their original access modifier when they become members of the subclass.

`protected` members

- A superclass' `private` members are hidden in its subclasses.
  - They can be accessed only through the `public` or `protected` methods inherited from the superclass.
- Subclass methods can refer to `public` and `protected` members inherited from the superclass simply by using the member names.
- When a subclass method overrides an inherited superclass method, the superclass method can be accessed from the subclass by preceding the superclass method name with keyword `super` and a dot (`.`) separator.

Access level modifiers

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Case study: A payroll application

- Suppose we need to create classes for two types of employees.
  - Commission employees are paid a percentage of their sales (`ComissionEmployee`).
  - Base-salaried commission employees receive a base salary plus a percentage of their sales (`BasePlusCommissionEmployee`).

Comparing the two types

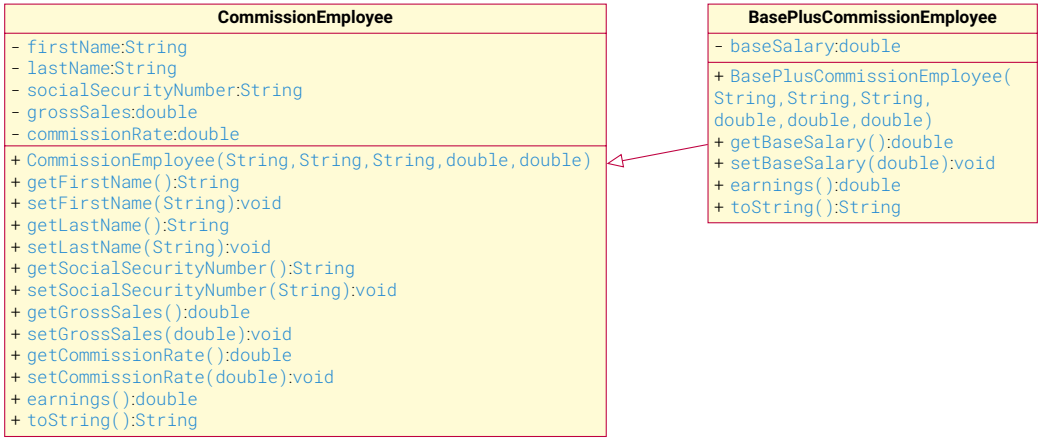
- Both classes need data fields to store the employee's personal information (e.g., name, ID card number).
- Differences
  - `BasePlusCommissionEmployee` class needs one additional field to store the employee's base salary.
  - The way of calculating the earnings are different.

Design Choice #1

- Two individual objects extending `Object`.
- Much of `BasePlusCommissionEmployee`'s code will be identical to that of `CommissionEmployee`.
- For implementation, we will literally copy the code of the class `CommissionEmployee` and paste the copied code into the class `BasePlusCommissionEmployee`.
  - "Copy-and-paste" approach is often error prone. it spreads copies of the same code throughout a system, creating code-maintenance nightmares.

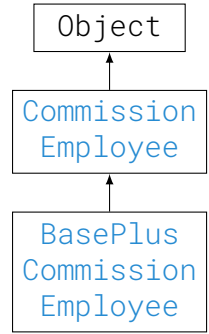
Design Choice #2

- `BasePlusCommissionEmployee` extends `CommissionEmployee`.



CommissionEmployee

```java
public class CommissionEmployee extends Object {
  private String firstName;
  private String lastName;
  private String socialSecurityNumber;
  private double grossSales;
  private double commissionRate;
  ...
}
```



CommissionEmployee

```java
public CommissionEmployee(String first, String last, String ssn,
                          double sales, double rate) {
  // implicit call to Object constructor occurs here
  firstName = first;
  lastName = last;
  socialSecurityNumber = ssn;
  setGrossSales(sales); // data validation
  setCommissionRate(rate); // data validation
}
// Several get and set methods
public void setGrossSales(double sales) {
  grossSales = (sales < 0.0) ? 0.0 : sales;
}
public void setCommissionRate(double rate) {
  commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
}
```

CommissionEmployee

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```
17 public void setFirstName(String first) {  firstName = first;  }
18 public String getFirstName() {  return firstName;  }
19 public void setLastName(String last) {  lastName = last;  }
20 public String getLastName() {  return lastName;  }
21 public void setSocialSecurityNumber(String ssn) { socialSecurityNumber =
       ssn; }
22 public String getSocialSecurityNumber() {  return socialSecurityNumber;
       }
23 public double getGrossSales() {  return grossSales;  }
24 public void setCommissionRate(double rate) {
25   commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
26 }
27 public double getCommissionRate() {  return commissionRate;  }
```

CommissionEmployee

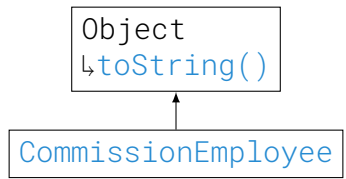
```
28 public double earnings() {
29   return commissionRate * grossSales;
30 }
31 @Override
32 public String toString() {
33   return String.format("%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
34     "commission employee", firstName, lastName,
35     "social security number", socialSecurityNumber,
36     "gross sales", grossSales,
37     "commission rate", commissionRate);
38 }
```

Inheriting `toString()` method

- A subclass inherits its superclass' methods
- Superclass' method can be called through a subclass object

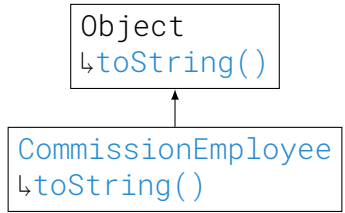
```
1 CommissionEmployee  obj = new CommissionEmployee();
2 obj.toString();
```

- If the subclass doesn't define a `toString()` method, `obj.toString()` is actually calling his superclass' `toString()` method.



Inheriting `toString()` method

- A subclass can override its superclass' method when it define a method with the same name.



Overriding `toString()` method

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

- `toString()` is one of the methods that every class inherits directly or indirectly from class `Object`.
  - Returns a String that "textually represents" an object.
  - Called implicitly whenever an object must be converted to a String representation (e.g., `System.out.println(objRef)`)
- Class `Object`'s `toString()` method returns a `String` that includes the name of the object's class.
  - If not overridden, returns something like "`CommissionEmployee@70dea4e`" (the part after `@` is the hexadecimal representation of the hash code of the object).
  - This is primarily a placeholder that can be overridden by a subclass to specify customized `String` representation.

Overriding `toString()` method

- To override a superclass' method, a subclass must declare a method with the same signature as the superclass method.
- `@Override` annotation
  - Optional, but helps the compiler to ensure that the method has the same signature as the one in the superclass.

BasePlusCommissionEmployee

```java
public class BasePlusCommissionEmployee extends CommissionEmployee {
  private double baseSalary;
  public BasePlusCommissionEmployee(
      String first, String last, String ssn,
      double sales, double rate, double salary) {
    super(first, last, ssn, sales, rate);
    setBaseSalary(salary);
  }
  public void setBaseSalary(double salary) {
    baseSalary = (salary < 0.0) ? 0.0 : salary;
  }
  ...
}
```

Subclass constructors

- Each constructor in the subclass needs to invoke a superclass constructor for object construction (e.g., to initialize inherited instance variables) by using the `super` keyword.
- If this is not explicitly done, the compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error.
- Constructor chaining: If a subclass constructor invokes a constructor of its superclass, there will be a whole chain of constructors called, all the way back to the constructor of Object. You need to be aware of its effect.

BasePlusCommissionEmployee

```java
@Override
public double earnings() {
  return baseSalary + ( commissionRate * grossSales );
}
@Override
public String toString() {
  return String.format("%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
    "base-salaried", firstName, lastName,
    "social security number", socialSecurityNumber,
    "gross sales", grossSales,
    "commission rate", commissionRate,
    "base salary", baseSalary);
}
```

Accessing fields of superclass

- What would happen if we compile the previous code?

```
BasePlusCommissionEmployee.java:46: error: commissionRate has private
 access in CommissionEmployee
      return baseSalary + (commissionRate * grossSales);
                           ^
BasePlusCommissionEmployee.java:46: error: grossSales has private
access in CommissionEmployee
      return baseSalary + (commissionRate * grossSales);
                                            ^
BasePlusCommissionEmployee.java:56: error: firstName has private
access in CommissionEmployee
      "base-salaried", firstName, lastName,
                       ^
```

Accessing fields of superclass

- A subclass inherits all `public` and `protected` members of its parent, no matter what package the subclass is in.
  - These members are directly accessible in the subclass
- If the subclass is in the same package as its parent, it also inherits the parent's *package-private* members (those without access level modifiers).
  - These members are directly accessible in the subclass
- A subclass does not inherit the `private` members. Private fields need to be accessed using the methods (`public`, `protected`, or package-private ones) inherited from superclass.

Solving the compilation problem

- Solution #1: using inherited methods.
- Solution #2: declaring superclass fields as `protected`.

More on the `super` keyword

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

- Two main usage scenarios:
- The `super` keyword can be used to invoke a superclass' constructor (as illustrated by our earlier example).
- If your method overrides its superclass' method, you can invoke the overridden method using the keyword `super`.

Referring to superclass method

```java
public class BasePlusCommissionEmployee extends CommissionEmployee {
  @Override
  public double earnings() {
    return baseSalary + ( getCommissionRate() * getGrossSales() );
  }
}
```

Referring to superclass method

```java
public class CommissionEmployee {
  public double earnings() {
    return commissionRate * grossSales;
  }
}
```

```java
public class BasePlusCommissionEmployee extends CommissionEmployee {
  @Override
  public double earnings() {
    return baseSalary + super.earnings();
  }
}
```

Inheritance in a nutshell

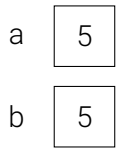
- The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some code you want, you can derive the new class from the existing one.
- The new class inherits its superclass' members – though the `private` superclass members are hidden in the subclass.

Inheritance in a nutshell

- You can customize the new class to meet your needs by including additional members and by overriding superclass members.
  - This does not require the subclass programmer to change (or even have access to) the superclass' source code.
  - Java simply requires access to the superclass' .class file.
- By doing this, you can reuse the *fields* and *methods* of the existing class without having to write (and debug!) them yourself.

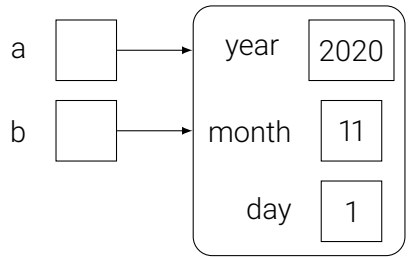
Copying primitive types

```
1 int a = 5;
2 int b = a;
```



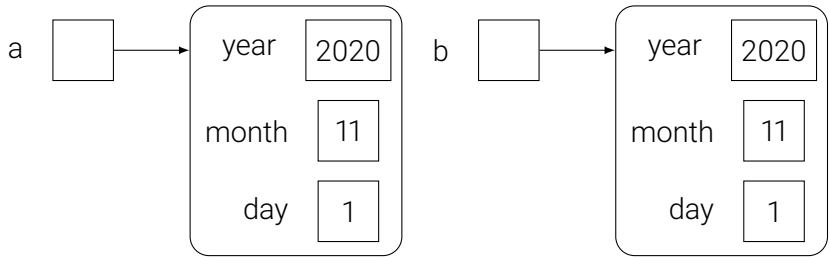
Copying reference types

```
1 Date a = new Date(2020,11,1);
2 Date b = a;
```



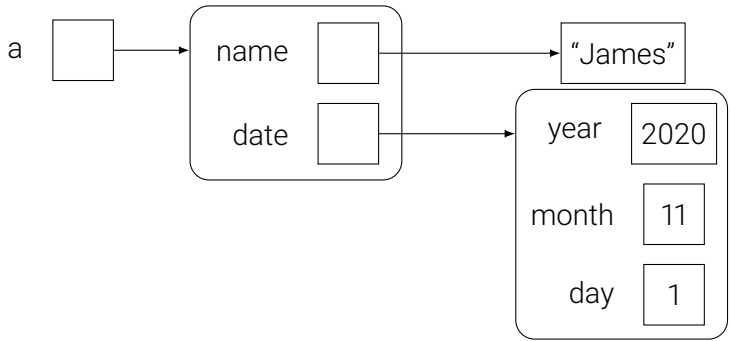
Copying reference types

```
1 Date a = new Date(2020,11,1);
2 Date b = a.clone();
```



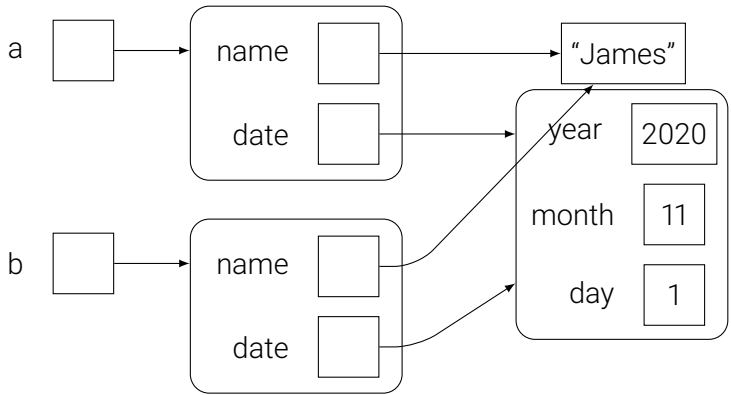
Copying reference types

```
1 Date d = new Date(2020,11,1);
2 Employee a = new Employee("James", d);
```



Copying reference types

```
1 Date d = new Date(2019,11,1);
2 Employee a = new Employee("James", d);
3 Employee b = a.clone();
```



Shallow copy

- `clone()` method makes a new object of same type as the original and copies all fields.
- However, if the fields are object references then original and cloned object can share common sub-objects.

Coding standards

Boss: “I think there might be some bugs in the code, please fix as soon as possible”

```
public class Permuter                                    {
    private static void permute(int n, char[] a)         {
        if (n == 0)                                      {
            System.out.println(String.valueOf(a))        ;}
        else                                             {
            for (int i = 0; i <= n; i++)                 {
                permute(n-1, a)                          ;
                swap(a, n % 2 == 0 ? i : 0, n)           ;}}}
    private static void swap(char[] a, int i, int j) {
        char saved = a[i]                                ;
        a[i] = a[j]                                      ;
        a[j] = saved                                     ;}}
```

Poor you: “Who wrote this piece of junk…”

What are coding standards?

- Coding standards are guidelines for code styles and documentation.
- The hope is that any developer familiar with the guidelines can work on any code that follows the guidelines.

Why coding standards are important?

- Coding standards lead to greater consistency within your code and the code of your teammates.
- Make code easier to understand.
- Make code easier to develop and maintain.
- Reduce overall cost (money, time, etc.) of project.

Basic coding standards for beginners

- Naming conventions
- Code formatting
- Comments and documentation

Naming conventions

- *Package* names should be in `lowercase`.

```
1 package edu.sustech.cs102
```

- *Class* names should be in *UpperCamelCase*.

```
1 public class HelloWorld {...}
```

- *Method* and *variable* names should be in `lowerCamelCase`.

```
1 public calculateTax() {...}
2 double monthlySalary = 333.3;
```

- *Constant* names should be in UPPER_CASE with words separated by "_".

```
1 static final double PLANK_CONSTANT = 6.62606896e-34;
```

Naming conventions

- What makes a good name?
  - Intention-revealing: `a()` vs. `computerTax()`.
  - Use full English descriptors: `firstName` vs. `fName`.
- Use terminologies applicable to the domain.
  - Business domain -`Customer`, software domain -`Client`.
- Use acronyms sparingly: `ssn` vs. `socialSecurityNumber`.
- Using mixed cases and underscores to make names readable.
- Avoid long names (generally, <15 characters is a good idea).

Formatting: Curly brace styles

- There is no "best" style that everyone should be following. The best style, is a consistent style. Styles also vary across languages.

```
if (condition) {
  doTask1();
} else {
  doTask2();
}
```

Most adopted (e.g., by Google), also suggested by Oracle.

```
if (condition)
{
  doTask1();
}
else
{
  doTask2();
}
```

Widely used.

```
if (condition)
{   doTask1();
}
else
{   doTask2();
}
```

Rarely used, but some people like it.

Formatting: Avoid deep nesting

```
if (cond1)) {
  if (cond2)) {
    if (cond3) {
      if (cond4)) {
        // ...
      } else {
        return false;
      }
  } else {
      return false;
    }
} else {
    return false;
  }
} else {
  return false;
}
```

```
if (!cond1) {
  return false;
}

if (!cond2) {
  return false;
}

if (!cond3) {
  return false;
}

if (cond4) {
  // ...
} else {
  return false;
}
```

Formatting: Code grouping

- Certain (sub)tasks often require a few lines of code.
- It is a good idea to keep these tasks within separate blocks of code, with some spaces between them.

```
while (bar > 0) {
  System.out.println();
  bar--;
}

if (oatmeal == tasty) {
  System.out.println("Oatmeal is good and good for you");
} else if (oatmeal == yak) {
  System.out.println("Oatmeal tastes like sawdust");
} else {
  System.out.println("tell me please what is this 'oatmeal'");
}
```

Formatting: Indentation

- Create indentations with spaces not tabs. A unit of indentation is usually 4 spaces.
- Why not tabs? Tab settings depend on editing environment.

```
while (bar > 0) {
  System.out.println();
  bar--;
}

if (oatmeal == tasty) {
  System.out.println("Oatmeal is good and good for you");
} else if (oatmeal == yak) {
  System.out.println("Oatmeal tastes like sawdust");
} else {
  System.out.println("tell me please what is this 'oatmeal'");
}
```

Formatting: Spaces

- Method names should be immediately followed by a left parenthesis.

```
foo (i, j); // NO!
foo(i, j); // YES!
```

- Array dereferences should be immediately followed by a left square bracket.

```
args [0];  // NO!
args[0];   // YES!
```

- Commas and semicolons are always followed by whitespace.

```
for (int i = 0;i < 10;i++)   // NO!
for (int i = 0; i < 10; i++) // YES!
getPancakes(syrupQuantity, butterQuantity);  // NO!
getPancakes(syrupQuantity, butterQuantity); // YES!
```

Formatting: Spaces

- Binary operators should have a space on either side.

```
a=b+c;          // NO!
a = b+c;        // NO!
a=b + c;        // NO!
a = b + c;      // YES!
z = 2*x + 3*y;          // NO!
z = 2 * x + 3 * y;      // YES!
z = (2 * x) + (3 * y);  // YES! Even better than the above one
```

Formatting: Spaces

- The keywords `if`, `while`, `for`, `switch`, and `catch` must be followed by a space.

```
if(hungry)  // NO!
if (hungry) // YES!
while(pancakes < 7)  // NO!
while (pancakes < 7) // YES!
for(int i = 0; i < 10; i++)  // NO!
for (int i = 0; i < 10; i++) // YES!
catch(TooManyPancakesException e)  // NO!
catch (TooManyPancakesException e) // YES!
```

Formatting: Class member ordering

```
class Order  {
  // fields

  // constructors

  // methods
}
```

Formatting: Maximum line length

- Avoid making lines longer than 120 characters. Most editors can easily handle 120 characters. Longer lines can be frustrating to work with.

Document your code

- Add concise and clear comments to explain
  - What a method does,
  - Method parameter and return values,
  - How methods modify objects,
  - Control structures,
  - Difficult or complex code (what it does and why code like this),
  - Processing order (or workflow).
- Avoid obvious comments.
- Learn more at `https://google.github.io/styleguide/javaguide.html`