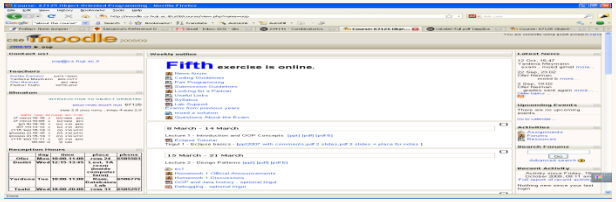
Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

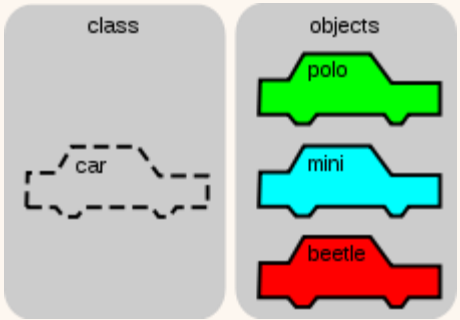
Lecture 1

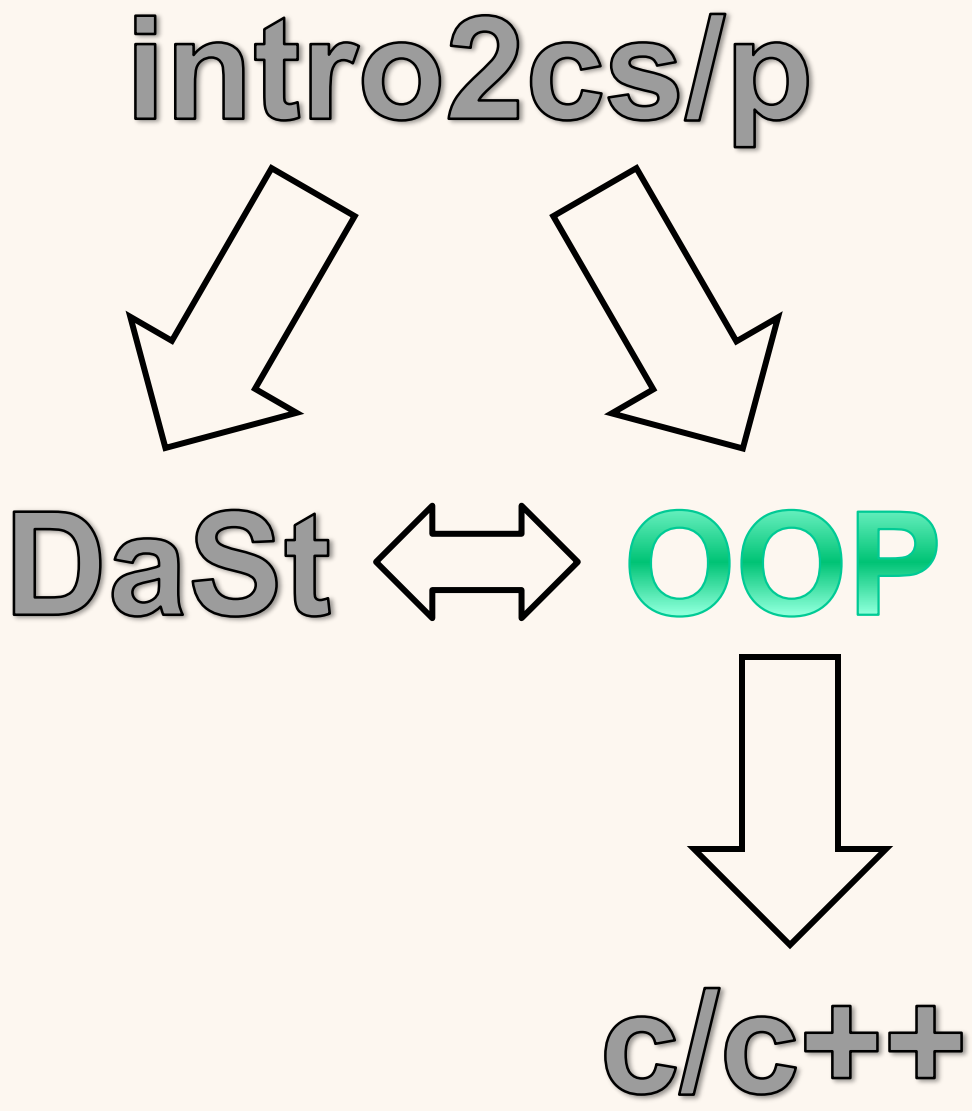
About the Course



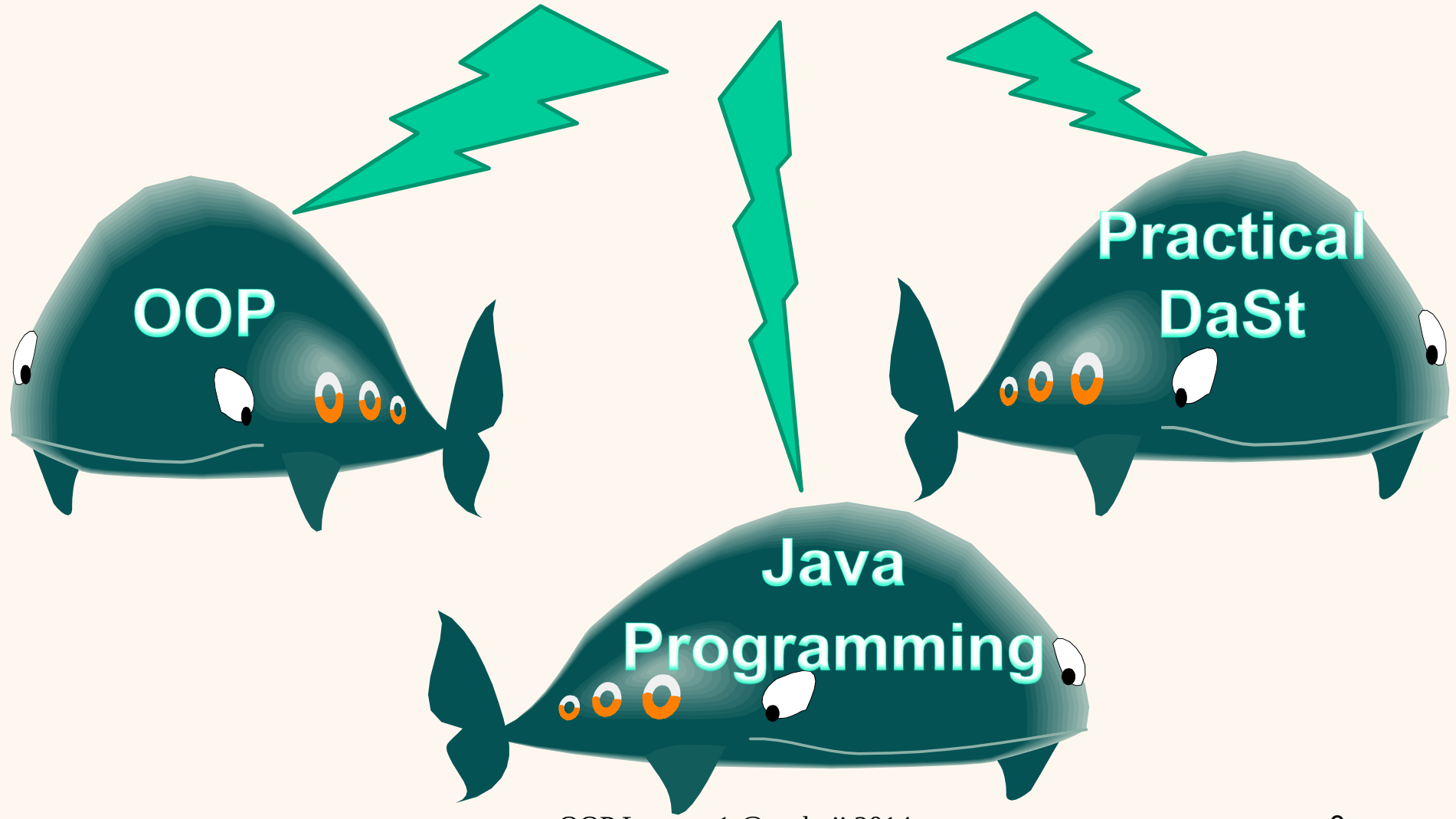
Object Oriented Programming Basics

Class vs. Object




intro2cs/p
DaSt
OOP
c/c++

Course Goals



Course Goals

- **Exercises**
  - Three basic java exercises
    - Ex1 will be released on **Wednesday**
  - Two implementations of a data structure or an algorithm taught in the **DaSt course**
  - Two **mini-projects** that require:
    - Complex **design**
    - Implementation of a **highly detailed system**
    - Testing your code

Same Name, Same Course?

- Both Intro2cs/p and this course underwent serious changes from recent years
- Students who passed intro2cs/p in 2012/2013 are exempted from ex1 and ex3
  - Upon providing proof of a passing grade
- The first five lectures and TAs are largely covered in the material taught in intro2cs/p in previous years
  - We rely on everything taught in this course in future slides, exercises and the final exam
  - We do recommend (at least) reviewing the slides for these lectures

Lectures

Lectures

Complementary Lectures

Sunday's Group only

- Friday, 14/3, 9:00-11:00
  - Alternative, attend Thursday's lesson (Thursday, 13/3, 14:00-16:00)

- Friday, 9/5, 10:00-12:00
  - Alternative, attend Thursday's lesson **the week before** (Thursday, 1/5, 14:00-16:00)

Course Staff

Junior Lecturers

Dan

Eyal

Inbar

Course Staff

Junior Lecturers

Dan

Eyal

Inbar

Teaching Assistants

Nir

Shachar

Course Staff

Junior Lecturers

Dan

Eyal

Inbar

Teaching Assistants

Nir

Shachar

Code Reviewers

Atara, Ayal, Ayelet, Itzhak, Liran, Nomi, Ofer, Rachel, Tamir

Properties of Good Program

User side…

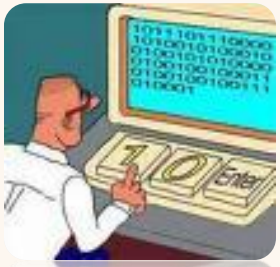
- Working!
- Meets the requirements
- Easy to learn and use
- Fast & efficient
- Fail-safe
- Fool-safe
- Hard-to-hack
- Compatible

Properties of Good Program

User side…

- Working!
- Meets the requirements
- Easy to learn and use
- Fast & efficient
- Fail-safe
- Fool-safe
- Hard-to-hack
- Compatible

Programmer side…



- Fast to code
- Easy to understand
  (by other team members or by same programmer in the future)
- Can be reused
- Easy to test
- Easy to debug
- Easy to update/upgrade

Basic OOP Concepts

1. Classes and Objects

- Easy to understand
- Easy to debug
- Fast-to-code

2. Inheritance / Hierarchy

3. Polymorphism

- Compatible
- Can be reused
- Easy to update/upgrade

4. Encapsulation / Information hiding

5. Genericity

- Hard-to-hack
- Can be reused
- Compatible

Basic OOD(esign)

1. Modularity

2. Design Patterns

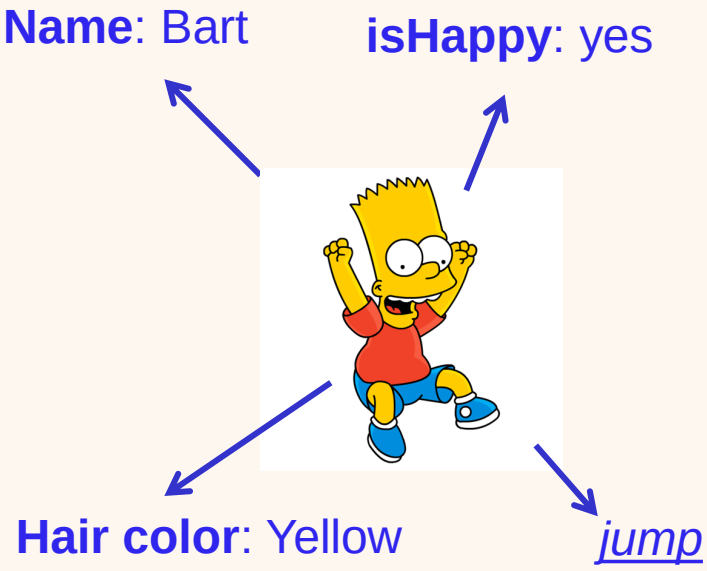
- Can be reused
- Easy to update/upgrade

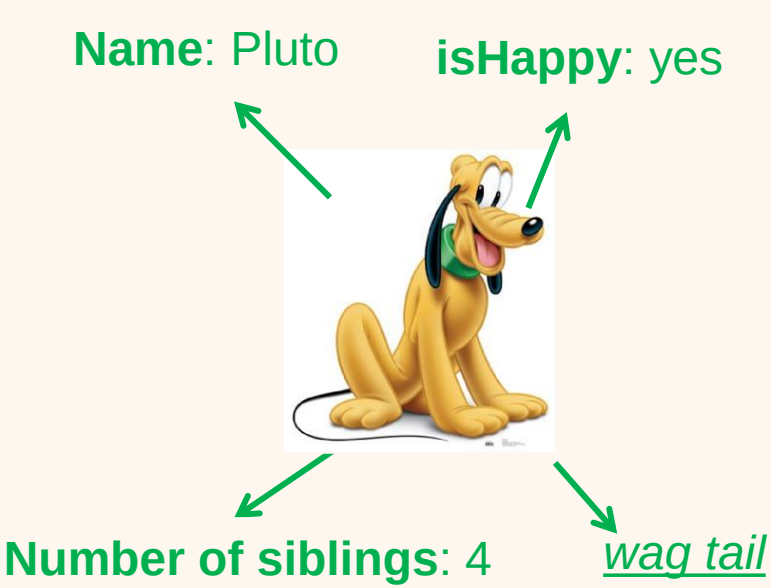
Object-Oriented Programming

- A programming paradigm, in which a program can be viewed as a set of interactions between **objects**
- An alternative to *procedural programming*
  - In which a program is a list of **procedures**
- Used in many programming languages
  - C++, PASCAL, Python
- The main programming principle in many languages
  - ***java***, C#

What is an Object?

- Real-world objects share two characteristics: They all have *state* and *behavior*

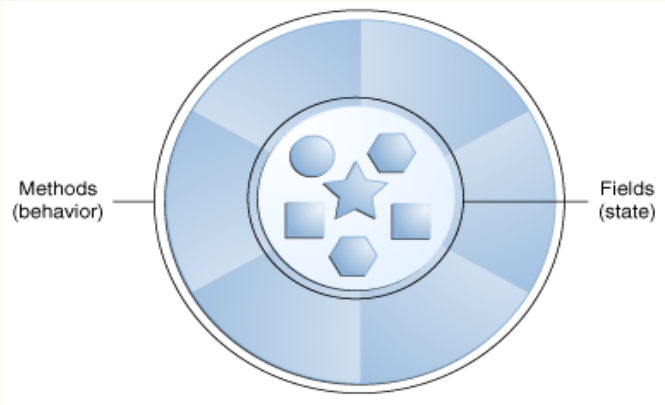




Software Objects

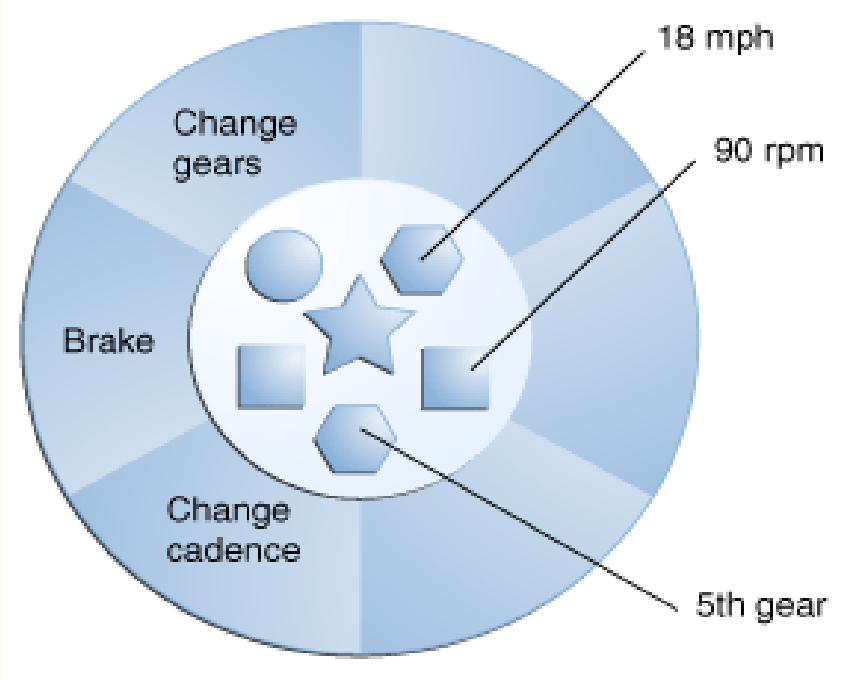
- Software objects can
  - Hold information (internal *states* – "*data members*")
  - Send messages to other objects (external *behavior* – "*methods*")

- Each object can be viewed as an independent "machine" with a distinct role or responsibility



Object Example

- A bicycle object



Object-oriented vs. Procedural Programming

- Procedural Programming
  - get_name(child)
  - wag_tail(dog)
  - get_length(string)
  - equals(dog1, dog2)

Object-oriented vs. Procedural Programming

- Procedural Programming
  - get_name(child)
  - wag_tail(dog)
  - get_length(string)
  - equals(dog1, dog2)
- Object-oriented Programming
  - child.getName()
  - dog.wagTail()
  - string.getLength()
  - dog1.equals(dog2)

OOP: Motivating Example
Animals

- Procedural Programming
  - bark(dog)
  - meow(cat)
  - moo(cow)
  - …

OOP: Motivating Example

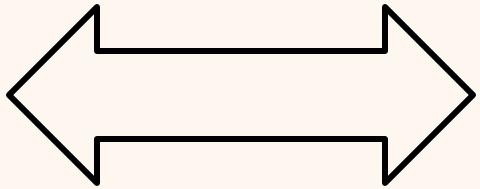
Animals

- Procedural Programming
  - bark(dog)
  - meow(cat)
  - moo(cow)
  - …
- Object-oriented Programming
  - dog.makeSound()
  - cat.makeSound()
  - cow.makeSound()

OOP: Motivating Example

Animals

- Procedural Programming
  - bark(dog)
  - meow(cat)
  - moo(cow)
  - …
- Object-oriented Programming
  - dog.makeSound()
  - cat.makeSound()
  - cow.makeSound()



Same Code

OOP: Motivating Example
Zoo

- But what about the code that calls these functions?
- Procedural Programming
  - Specific code for each case
  - Changes and extensions are **hard** and **bug prone**

OOP: Motivating Example
Zoo

- But what about the code that calls these functions?

- Procedural Programming
  - Specific code for each case
  - Changes and extensions are **hard** and **bug prone**

- Object-oriented Programming
  - General code (all use the **same method name**)
  - Easy to use
  - Easy to extend

Some Objects are Similar

- In the real world, many individual objects are of the same kind
  - Many different dogs exist
  - Each dog has 4 logs, can wag its tail, etc.
- However, different objects of the same kind may have different states
  - Pluto, Guffy, and Rex are all different dogs

Classes

- Software class are used to define groups of objects
- These groups share the same *types of members* (i.e., possible *states*) and the same *methods* (i.e., *behavior*)
- Objects of a given class (denoted *instances*) give concrete values to each of the data members
  - Goofy (a dog *object*) is an *instance* of the class Dog
  - myBike is an *instance* of the class Bicycle

Data Members

- Data members are variables that belong to a specific object

- In java, each data member has a type
  - Primitives: **int**, **double**, **boolean**, …
  - Other classes: String, Dog, Bicycle, …

Methods

- Methods are functions that are associated with a specific class
- Methods can access the data members of a given object

Java Methods

- Java methods can get a set of parameters
  - Each parameter has its own type

Java Methods

- Java methods can get a set of parameters
  - Each parameter has its own type
- Java methods can return a value of a given type
  - This type can be either primitive or another class
  - To specify the type, you write it before the function name
  - The return statement is done using the **return** keyword

Java Methods

- Java methods can get a set of parameters
  - Each parameter has its own type
- Java methods can return a value of a given type
  - This type can be either primitive or another class
  - To specify the type, you write it before the function name
  - The return statement is done using the **return** keyword
- Methods can also return nothing
  - Instead of type, use the **void** keyword

Java Methods

- Java methods can get a set of parameters
  - Each parameter has its own type
- Java methods can return a value of a given type
  - This type can be either primitive or another class
  - To specify the type, you write it before the function name
  - The return statement is done using the **return** keyword
- Methods can also return nothing
  - Instead of type, use the **void** keyword
  - Does this look familiar?

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

Java Methods

- Java methods can get a set of parameters
  - Each parameter has its own type
- Java methods can return a value of a given type
  - This type can be either primitive or another class
  - To specify the type, you write it before the function name
  - The return statement is done using the **return** keyword
- Methods can also return nothing
  - Instead of type, use the **void** keyword
  - Does this look familiar?

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

Java Names

- java **classes** should be a sequence of on or more words, all starting with a capital Letters, followed by lower-case letters
  - String, MyInteger, WhatAnExcellentClass, …
- java **methods** and **members** follow the same rules, but **start with a lower-case letter**
  - print(), myMember, superbFunction(…), …

Class Example

Bicycle.java

```
class Bicycle {
    /* Data members */
    int speed = 0;
    int gear = 1;

    // Methods
    void changeGear(int newValue) {
        gear = newValue;
    }
    int speedUp(int increment) {
        speed = speed + increment;
        return speed;
    }

    void stop() {
        speed = 0;
        return;
    }
    …
}
```

Class Example

Bicycle.java

```
class Bicycle {
    /* Data members */
    int speed = 0;
    int gear = 1;

    // Methods
    void changeGear(int newValue) {
        gear = newValue;
    }
    int speedUp(int increment) {
        speed = speed + increment;
        return speed;
    }

    void stop() {
        speed = 0;
        return;
    }
    …
}
```

Class starts here

Class Example

Bicycle.java

```
class Bicycle {
    /* Data members */
    int speed = 0;
    int gear = 1;

    // Methods
    void changeGear(int newValue) {
        gear = newValue;
    }

    int speedUp(int increment) {
        speed = speed + increment;
        return speed;
    }

    void stop() {
        speed = 0;
        return;
    }
    …
}
```

Class starts here

Class ends here

Class Example

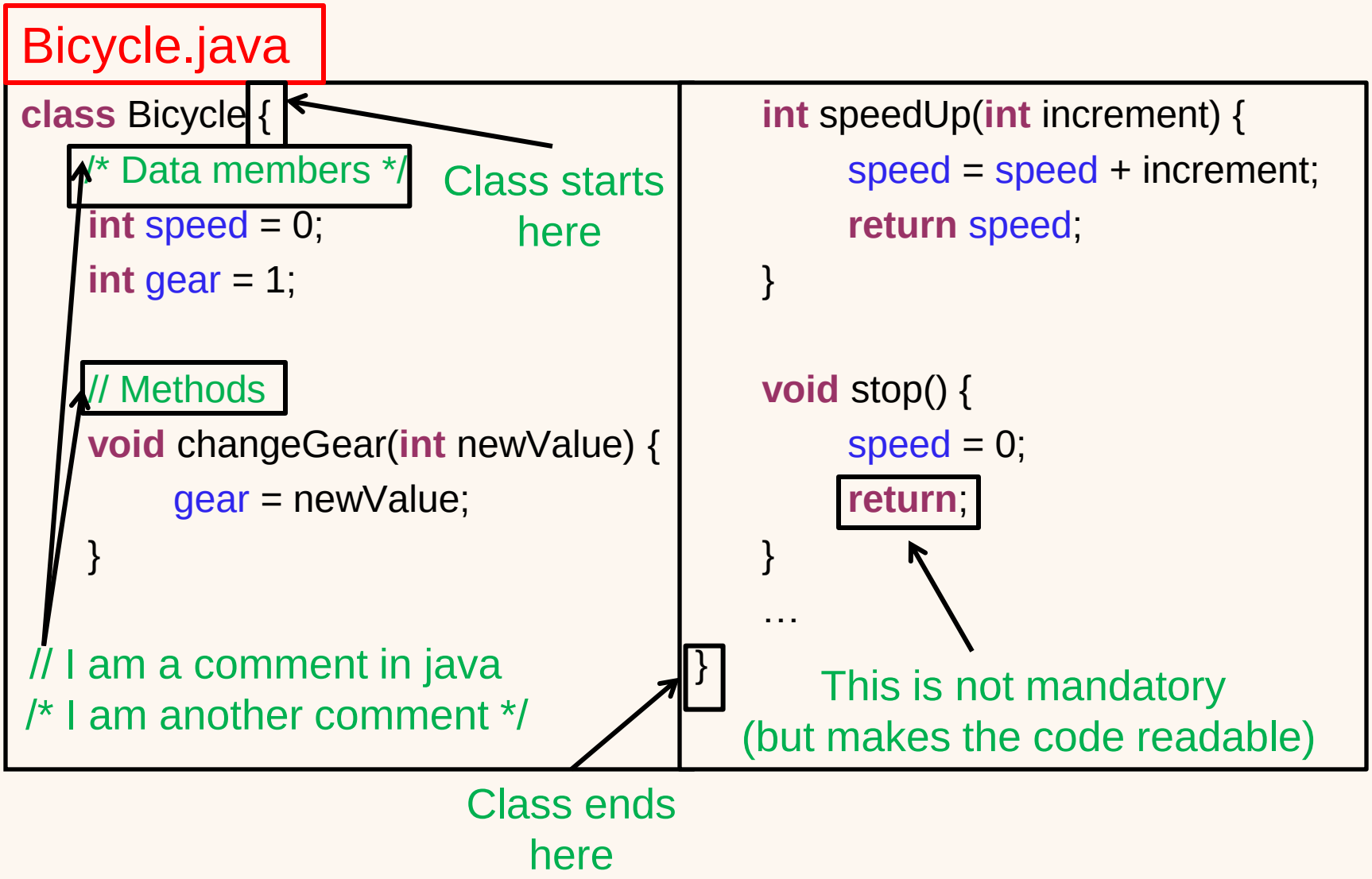
Bicycle.java

```
class Bicycle {
    /* Data members */
    int speed = 0;
    int gear = 1;

    // Methods
    void changeGear(int newValue) {
        gear = newValue;
    }

    int speedUp(int increment) {
        speed = speed + increment;
        return speed;
    }

    void stop() {
        speed = 0;
        return;
    }
    …
}

// I am a comment in java
/* I am another comment */
```

Class starts here

Class ends here

Class Example



Class and Objects

- Every instance of the bicycle class has the same types of members (gear and speed) and the same methods (changeGear(), speedUp() and stop())

- Every instance has its own *state* (i.e., different values for the different members)
  - Two different Bicycle instances may share the same values to the same members

Creating New Objects

- Classes define how each of their objects (instances) look like
  - What are their members and methods
- Each class defines a special method (or methods) called *constructor(s)* that allow the creation of new objects

Constructors

- Constructors are methods used for assigning values to the data members of the new object

- Java constructors have several properties:
  - They use the same name as the class
  - They have no return value
  - They can get a set of parameters, just like any other method (including no parameters)

Constructor Example

Bicycle.java

```
class Bicycle {
        /* Data members */
      int speed = 0;
      int gear;

        /* Constructors */
      Bicycle(int newGear) {
            gear = newGear;
      }

        /* Methods */
      void changeGear(int newValue) {
            gear = newValue;
      }
      int speedUp(int increment) {
            speed = speed + increment;
            return speed;
      }

      void stop() {
            speed = 0;
            return;
      }
      …
}
```

Constructor Example

Bicycle.java

```
class Bicycle {
        /* Data members */
      int speed = 0;
      int gear;

        /* Constructors */
      Bicycle(int newGear) {
            gear = newGear;
      }

        /* Methods */
      void changeGear(int newValue) {
            gear = newValue;
      }
      int speedUp(int increment) {
            speed = speed + increment;
            return speed;
      }

      void stop() {
            speed = 0;
            return;
      }
      …
}
```

Using Constructors

BicycleDemo.java

```
class BicycleDemo {
        public static void main(String[] args) {
                // Create two different Bicycle objects
                Bicycle bike1 = new Bicycle(1);
                Bicycle bike2 = new Bicycle(2);




        }
}
```

Output:

Using Constructors

BicycleDemo.java

```
class BicycleDemo {
        public static void main(String[] args) {
                // Create two different Bicycle objects
                Bicycle bike1 = new Bicycle(1);
                Bicycle bike2 = new Bicycle(2);

                // Invoke methods on those objects
                bike1.speedUp(10);
                bike1.changeGear(2);
                System.out.println(bike1.gear+“, ”+bike1.speed);



        }
}
```

Output:

Using Constructors

BicycleDemo.java

```
class BicycleDemo {
        public static void main(String[] args) {
                // Create two different Bicycle objects
                Bicycle bike1 = new Bicycle(1);
                Bicycle bike2 = new Bicycle(2);

                // Invoke methods on those objects
                bike1.speedUp(10);
                bike1.changeGear(2);
                System.out.println(bike1.gear+“, ”+bike1.speed);

        }
}
```

Output:
2, 10

Using Constructors

BicycleDemo.java

```
class BicycleDemo {
        public static void main(String[] args) {
                // Create two different Bicycle objects
                Bicycle bike1 = new Bicycle(1);
                Bicycle bike2 = new Bicycle(2);

                // Invoke methods on those objects
                bike1.speedUp(10);
                bike1.changeGear(2);
                System.out.println(bike1.gear+", "+bike1.speed);
                bike1.changeGear(3);
                System.out.println(bike1.gear+", "+bike1.speed);



        }
}
```

Output:
2, 10

Using Constructors

BicycleDemo.java

```
class BicycleDemo {
        public static void main(String[] args) {
                // Create two different Bicycle objects
                Bicycle bike1 = new Bicycle(1);
                Bicycle bike2 = new Bicycle(2);

                // Invoke methods on those objects
                bike1.speedUp(10);
                bike1.changeGear(2);
                System.out.println(bike1.gear+", "+bike1.speed);
                bike1.changeGear(3);
                System.out.println(bike1.gear+", "+bike1.speed);



        }
}
```

Output:
2, 10
3, 10

Using Constructors

BicycleDemo.java

```
class BicycleDemo {
        public static void main(String[] args) {
                // Create two different Bicycle objects
                Bicycle bike1 = new Bicycle(1);
                Bicycle bike2 = new Bicycle(2);

                // Invoke methods on those objects
                bike1.speedUp(10);
                bike1.changeGear(2);
                System.out.println(bike1.gear+", "+bike1.speed);
                bike1.changeGear(3);
                System.out.println(bike1.gear+", "+bike1.speed);
                bike2.speedUp(10);
                bike2.stop();
                System.out.println(bike2.gear+", "+bike2.speed);
        }
}
```

Output:
2, 10
3, 10

Using Constructors

BicycleDemo.java

```
class BicycleDemo {
        public static void main(String[] args) {
                // Create two different Bicycle objects
                Bicycle bike1 = new Bicycle(1);
                Bicycle bike2 = new Bicycle(2);

                // Invoke methods on those objects
                bike1.speedUp(10);
                bike1.changeGear(2);
                System.out.println(bike1.gear+", "+bike1.speed);
                bike1.changeGear(3);
                System.out.println(bike1.gear+", "+bike1.speed);
                bike2.speedUp(10);
                bike2.stop();
                System.out.println(bike2.gear+", "+bike2.speed);
        }
}
```

Output:
2, 10
3, 10
2, 0

The String Class

- The most common java class
  - Represent a string of characters
- Can be initialized using the “=“ sign
  - String myString = “hello”;
- Has many important methods
  - length(), charAt(…),
- Is immutable
  - Can’t change content of string (e.g., change 1st char to ‘y’)
  - More on this to come

Classes vs. Objects
Recursive Structure

- A class can hold data members of its own type
  - This is quite common
  - Example: A Dog can have a “mother” member (of type Dog)

Recursive Structure Example

Dog.java

```
class Dog {
        /* Data members */
        String name;
        Dog mother;

        /* Constructors */
        Dog (String dogName, Dog dogMother) {
                name = dogName;
                mother = dogMother;
        }

        …
}
```

Classes vs. Objects
Memory Issues

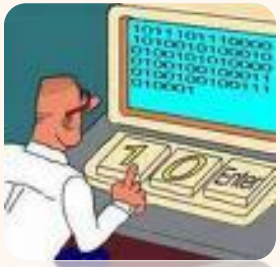
- Only one copy of the class exist
  - Memory to store methods is only allocated once
- For each class, many objects potentially exist
  - Memory is allocated for each of them

Reminder: Properties of Good Program

User side…

- Working!
- Meets the requirements
- Easy to learn and use
- Fast & efficient
- Fail-safe
- Fool-safe
- Hard-to-hack
- Compatible

Programmer side…



- Fast to code
- Easy to understand
  (by other team members or by same programmer in the future)
- Can be reused
- Easy to test
- Easy to debug
- Easy to update/upgrade

Introduction to Object-oriented Design

- TMTOWTDI
  - There's more than one way to do it
- A good software solution to a problem is a solution that yields a **good program**
  - Working, extensible, easy to debug, efficient, etc.
- There is hardly ever a perfect solution
  - A good design is one in which the pros out-weight the cons
  - Obtaining one usually requires expertise

Constants

- Many programming languages (including *java*) allow the creation of constant variables
  - These are variables that never change
  - Their value is set once at the creation of the object

- In java, you add the keyword **final** before the member type

Why Use Constants?

- Some properties of an object should never be changed
  - A dog's name
  - A bike's maximal gear
- We should decide, at design time, which properties (i.e., data members) should remain the same throughout the lifetime of the object

Constants Example

Dog.java

```
class Dog {
        /* Data Members*/
        final String name;
        Dog mother;

        /* Constructors */
        Dog (String dogName, Dog dogMother) {
                name = dogName;
                mother = dogMother;
        }

        …
        public static void main(String args[]) {
                Dog myDog = new Dog(“pluto”, otherDog);
                myDog.name = “goofy”;          // Compilation error
        }
}
```

Why Force It?

- If someone wants to change a dog's name, why should we prevent her from doing it?
- A major issue in design is prevention instead of cure
  - If we design a dog class such that its name should never change, we should **prevent users from changing it**
  - Users can be either us, or other programmers that use our code
- When someone uses our code in the wrong way, **bugs** occurs
  - This may not be our fault, but this is **our problem**

Types

- Each java variable can either be
  - a primitive (int, double, char…)
  - or a reference (to an object)

Reference vs. Content

- The following line contains two parts, separated by the “=” sign:

Bicycle bike1 = ***new*** *Bicycle(1);*

- The first part (*Bicycle bike1)* defines a new reference to an object of type *Bicycle*

- The second part (***new*** *Bicycle(1)*) defines content
  - A concrete object

Reference

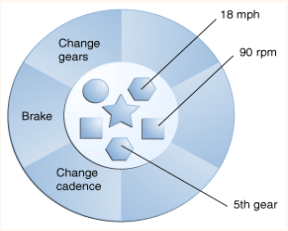
- A reference is not an actual object, but something that **points** to an object

- This means that creating new references doesn't waste much memory
  - *Bicycle bike1 = **new** Bicycle(1);*
  - *Bicycle bike2 = bike1;*
  - *Bicycle bike3 = bike1;*
  - *…*

Reference

- A reference is not an actual object, but something that **points** to an object

- This means that creating new references doesn't waste much memory
  - *Bicycle bike1 = **new** Bicycle(1);*
  - *Bicycle bike2 = bike1;*
  - *Bicycle bike3 = bike1;*
  - *…*

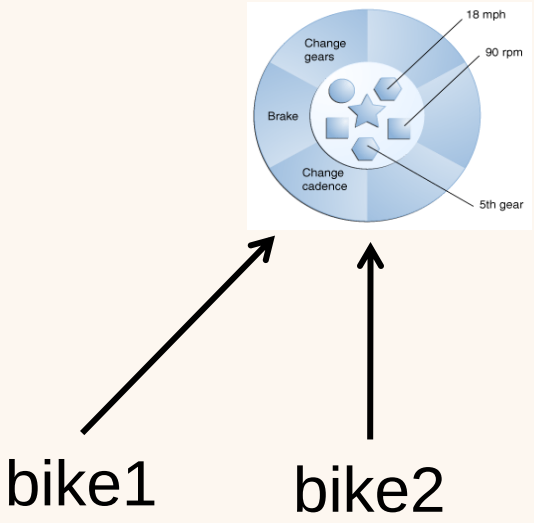


bike1

Reference

- A reference is not an actual object, but something that **points** to an object

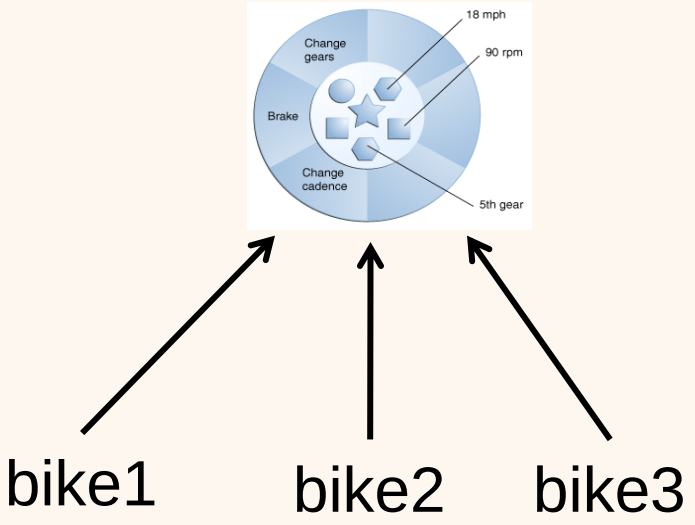
- This means that creating new references doesn't waste much memory
  - *Bicycle bike1 = **new** Bicycle(1);*
  - *Bicycle bike2 = bike1;*
  - *Bicycle bike3 = bike1;*
  - *…*



Reference

- A reference is not an actual object, but something that **points** to an object

- This means that creating new references doesn't waste much memory
  - *Bicycle bike1 = **new** Bicycle(1);*
  - *Bicycle bike2 = bike1;*
  - *Bicycle bike3 = bike1;*
  - *…*



Content

- Calling a constructor (using the **new** keyword) creates a new object
  - Each call requires more memory

Content Example

*Bicycle bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

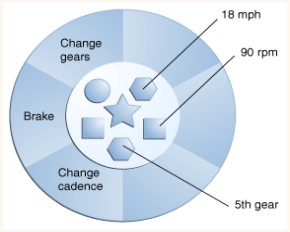
*bike1 = **new** Bicycle(1);*

Content Example

*Bicycle bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*



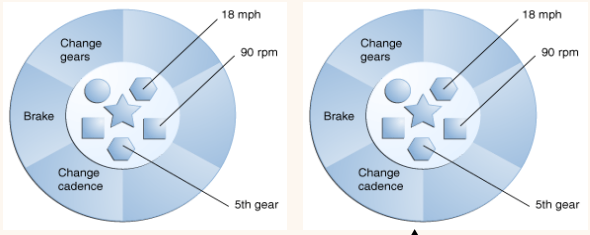
bike1

Content Example

*Bicycle bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*



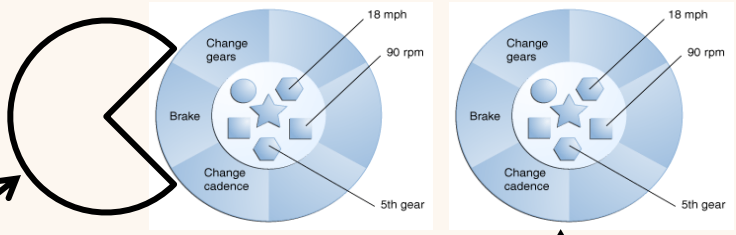
bike1

Content Example

*Bicycle bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*



Garbage Collector

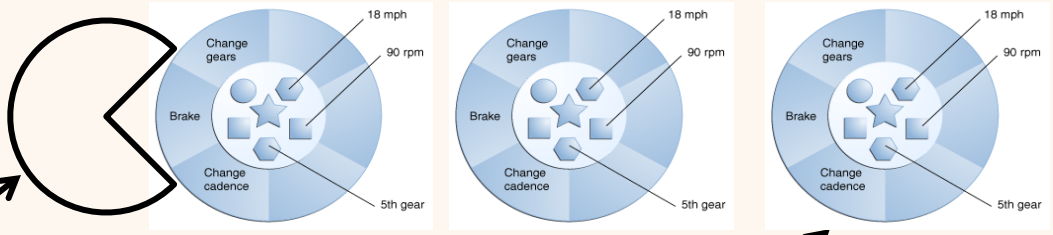
bike1

Content Example

*Bicycle bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*



Garbage Collector

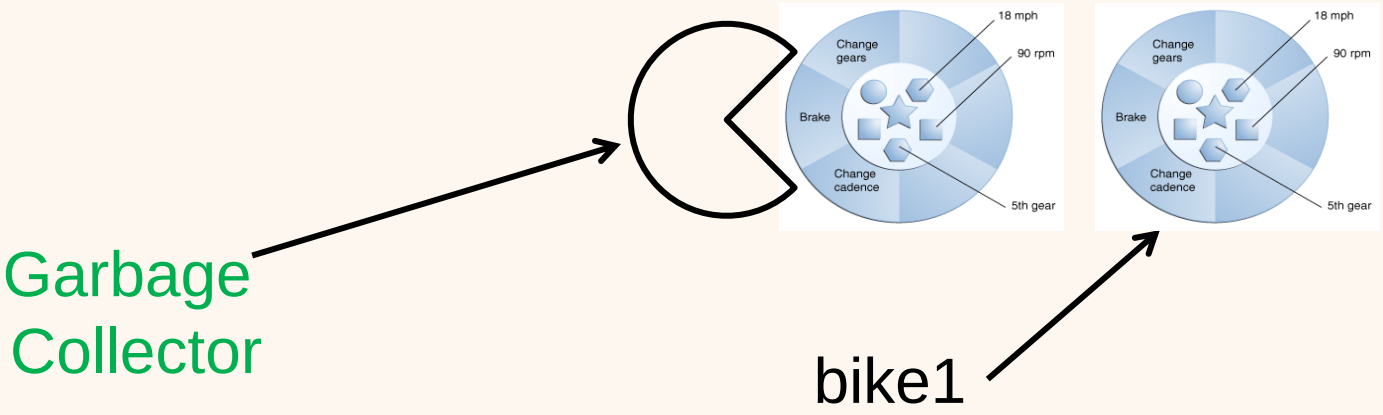
bike1

Content Example

*Bicycle bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

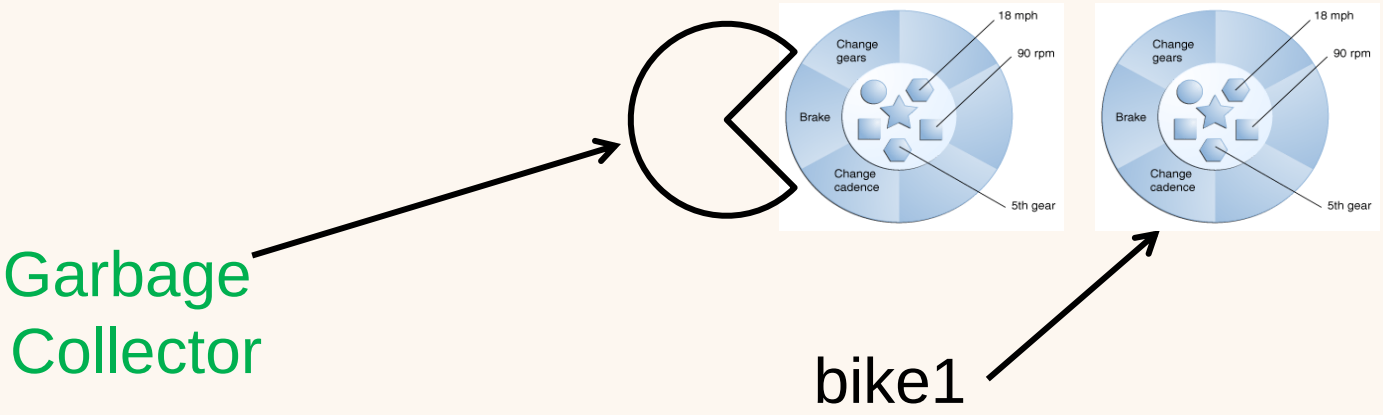


Content Example

*Bicycle bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*



- The reference-content distinction has other implications
  - See later in this course

Primitives

- A primitive variable contains data
  - `int num = 5;` // num now holds the number 5

Primitives

- A primitive variable contains data
  - `int num = 5;` // num now holds the number 5
  - `int num2 = myNum;` // num2 also holds the number 5

Primitives

- A primitive variable contains data
  - `int num = 5;` // num now holds the number 5
  - `int num2 = myNum;` // num2 also holds the number 5
  - `num = 6;` // num now holds the number 6
    // num2 **still** holds the number 5

Primitives

- A primitive variable contains data
  - **int** num = 5; // num now holds the number 5
  - **int** num2 = myNum; // num2 also holds the number 5
  - num = 6; // num now holds the number 6
    // num2 **still** holds the number 5

- Each **int** requires the same amount of memory

So far…

- Writing a Good Program
  - Works, fast, extensible, …
- Object-oriented Programming
  - Class vs. object
  - Constructors
  - Constants
  - Reference vs. content