# More OO Examples

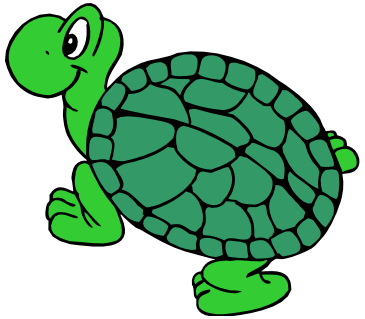
covering

** steps to write OO programs
** overloading ** overriding

Chapter 2 (sections 2.1–2.3; 2.5–2.8; 2.10–2.11) – “Big Java” book
Chapters 2-4 – “Head First Java” book
Chapters 4+6 – “Introduction to Java Programming” book
Chapter 3 – “Java in a Nutshell” book

# Example 1

- Sam is a turtle.
- Sam is **green** with a scaly tail and a top speed of 2 miles per hour.
- Peter is a rabbit. Peter is **grey**, has rough fur, a fluffy tail and can run at a speed of 150 miles per hour.
- Both rabbits and turtles can swim, but only rabbits sleep.



Objects – classes/instances?
Attributes?
Operations?

# Class Diagrams

For the Rabbit?

| Rabbit |
| --- |
| String name;<br>String tailType;<br>Color color;<br>int speed;<br>String furType; |
| run();<br>sleep();<br>swim(); |

For the Turtle?

| Turtle |
| --- |
| |
| |

# Rabbit: Class Definition and Information Hiding

```
import java.awt.*;
/**
 * Title:        Rabbit.java
 * Description: This class contains the definition of a rabbit.
 * Copyright:   Copyright (c) 2001
 * @author      Laurissa Tokarchuk
 * @version     1.0
 */
public class Rabbit {
  // Declaration of instance variables
  String name, tailType, furType;
  Color color;
  int   speed;

  // Declaration of methods – blank for now
}
```

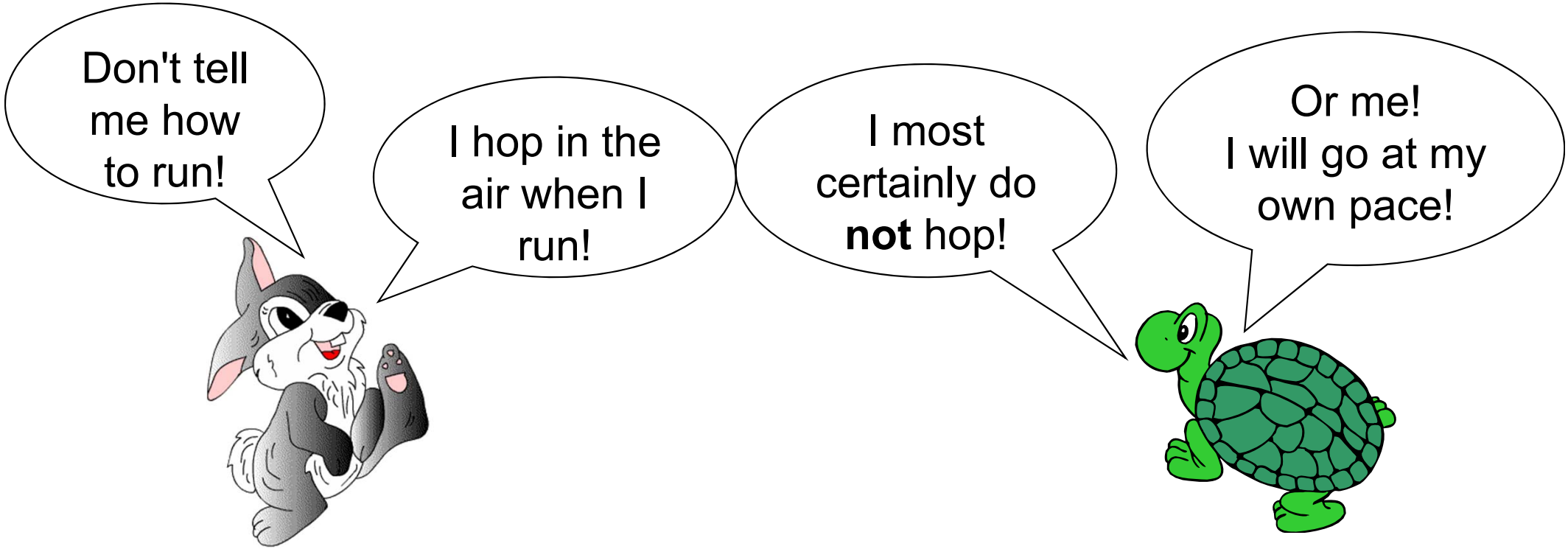
| Rabbit |
| --- |
| String name;<br>String tailType;<br>Color color;<br>int speed;<br>String furType; |
| run();<br>sleep();<br>swim(); |

```
public class Rabbit {
  private String name, tailType, furType;
  private Color color;
  private int   speed;
}
```

Queen Mary University of London

# Controlling our creatures …

This slide has lots of animation; you must be in class to fully understand.

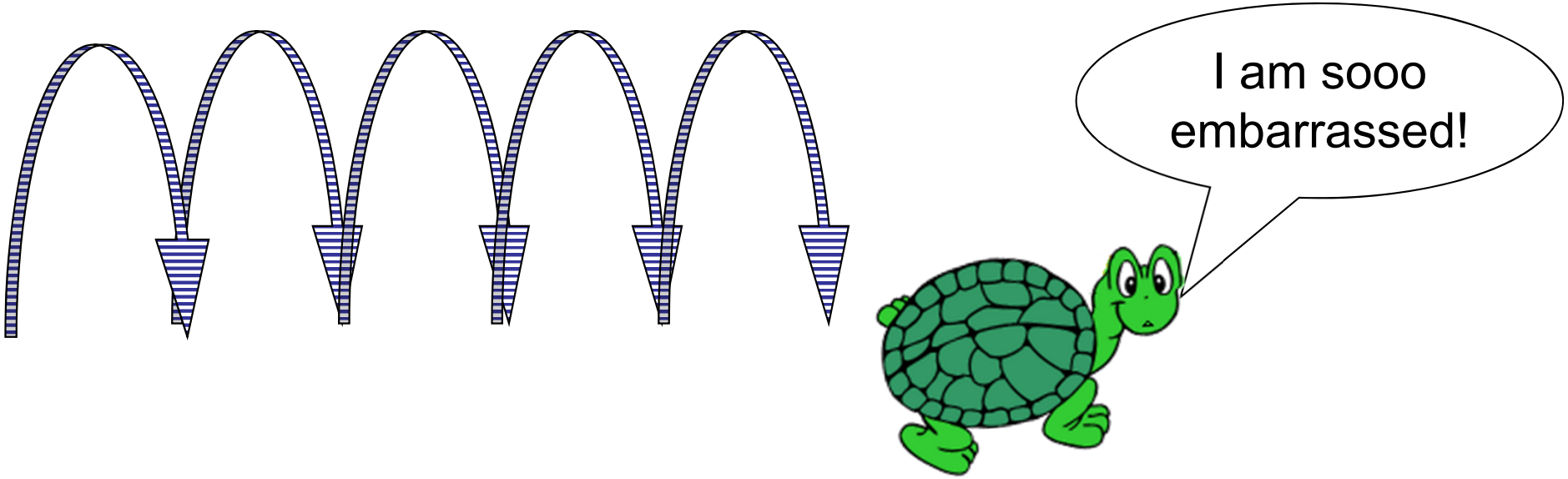



Rabbits and Turtles provide an interface for a client to request that they run. But the client has no control over how they do that!

# Without Data Hiding

This slide has lots of animation; you must be in class to fully understand.

- If the client could control the creature objects, it could make our poor turtle hop! ☹




We need to make sure that rabbits and turtles have the control they require!

# Accessor and Mutator methods for rabbit

```
/**
 * This method gets the furType of the rabbit.
 * @return String  Type of fur.
 */
public String getFurType() {
   return furType;
}
```

```
    /**
     * This method sets the furType of the rabbit.
     * @param furtype  Type of fur the rabbit should have.
     */
    public void setFurType(String furType) {
      // check to see that the furType is valid for rabbits
      if ((furType.equals("scaly") || (furType.equals("bald")) {
        System.out.println("ERROR: Illegal fur type.");
      }
      else this.furType = furType;
    }
```

# Rabbit Methods

```
/**
  * This is the sleep method for the rabbit. It dictates the
  * number of minutes the rabbit sleeps.
  * @param duration  The number of minutes to sleep.
  */
public void sleep(int duration) {
   // Code of sleep
}
```

```
/**
  * This method allows the rabbit to run. The distance the
  * rabbit runs depends on how long the rabbit runs for, and
  * on whether or not it is running in a zigzag.
  * @param duration  The number of minutes to run.
  * @param zigzag    Whether to run in a zigzag pattern
  * @return int      Number of miles run..
  */
public int run(int duration, boolean  zigzag) {
  // code of run
}
```

# Method overloading

- Whenever two or more methods have the same name but different input parameters. For example,

```
public int run(int duration,  boolean  zigzag) { }
public int run(int duration){ }
```

- Both of these methods can exist in the class **Rabbit**.
  - Which one is called depends on how you call it, e.g.

```
Rabbit bugs = new Rabbit();
int distance = bugs.run(5, true);  // OR
int distance2 = bugs.run(5);
```

# Rabbit class (in full)

```
import java.awt.*;
public class Rabbit {
  private String name, tailType, furType;
  private Color color;
  private int   speed;

  public String getFurType() { return furType; }

  public void setFurType(String furType) {
    // check to see that the furType is valid for rabbits
    if ((furType.equals("scaley") || (furType.equals("bald")) {
      System.out.println("ERROR: Illegal fur type.");
    }
    else this.furType = furType;
  }
  public void sleep(int duration) {
    // code of sleep
  }
  public int run(int duration,  boolean  zigzag){
    // code of run
  }
  public void swim() {
    // code of swim
  }
}
```

You should always write full comments in the program. Some comments have been removed here to save space on the slide.

# Writing our Test class …

```
public class RabbitTest {
  public static void main(String[] args) {
    Rabbit bunny = new Rabbit();
  }
}
```

What are the values of **`name`**, **`furType`** and **`speed`**?

All instance variables are set to their default values, unless otherwise specified.

# Initialisation and Constructors

- In Java, all variables must be initialised before they can be used.
- Java automatically sets some initial values for you for variables of the class (instance variables), but not for variables in methods.

| field type | initial value |
|---|---|
| boolean | false |
| char | '\u0000' |
| byte, short, int, long | 0 |
| float / double | +0.0f / +0.0d |
| object reference | null |

# Using Objects

- So by default, our **`Rabbit`** class has a constructor provided by Java. Thus we can create a **`Rabbit`** object as follows:

```
Rabbit  bunny;          // bunny is 'null' now.
bunny = new Rabbit(); // creates a rabbit
```

or equivalently:

```
Rabbit bunny = new Rabbit();
```

A **`Rabbit`** object (or any object) is a reference variable.
This is different from a primitive variable (e.g. **`int`**, **`double`**).

# Rabbit's creation … (1/2)

This slide has lots of animation; you must be in class to fully understand.

```
Rabbit bunnie = new Rabbit();
```


**Rabbit**

Assigns the new **Rabbit** object to the reference variable **bunnie**

**Rabbit** object

Tells the JVM to allocate space for a reference variable of type **Rabbit** (forever) called **bunnie**

Tells the JVM to allocate space for a new **Rabbit** object on the heap

# Rabbit's creation … (2/2)

This slide has lots of animation; you must be in class to fully understand.


Now they are joined!

– The variable **`bunnie`** now controls the **`Rabbit`** object.

```
bunnie.sleep(5);
```

# Life on the Heap! (1/2)

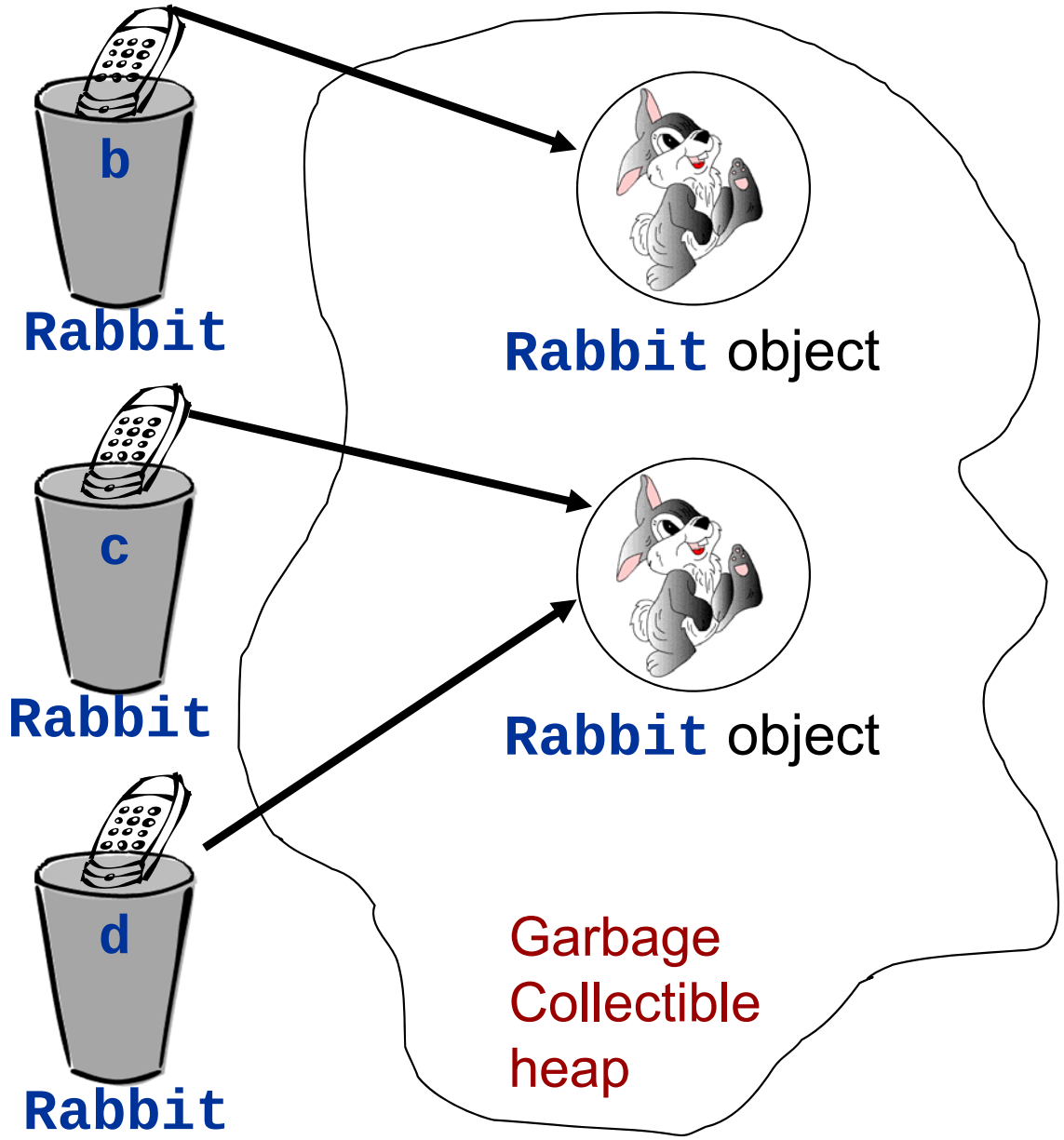
This slide has lots of animation; you must be in class to fully understand.

```
Rabbit b = new Rabbit();

Rabbit c = new Rabbit();

// d and c are different
// variables, that refer
// to the SAME object.

Rabbit d = c;
```



# Life on the Heap! (2/2)

This slide has lots of animation; you must be in class to fully understand.

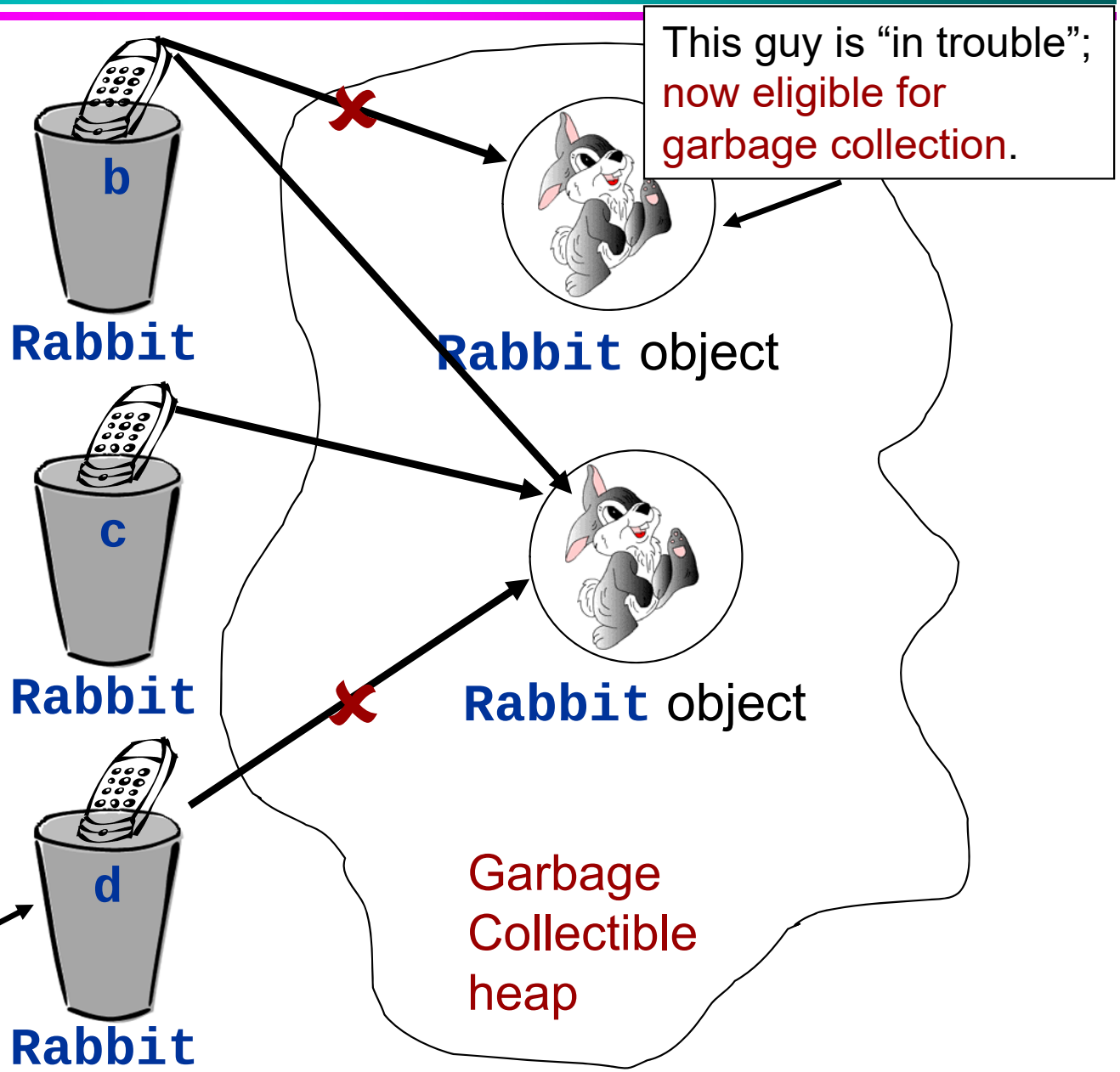
```
Rabbit b = new Rabbit();
Rabbit c = new Rabbit();

// d and c are different
// variables, that refer
// to the SAME object.

Rabbit d = c;

b = c;

d = null;
```



# Common Error


**Rabbit**

```
Rabbit bugs;
bugs.sleep(5);
```

- The **bugs** rabbit cannot sleep!
- In fact, any attempt to use **bugs** will result in:

```
Exception in thread "main"
  java.lang.NullPointerException
```

Don't try to use reference variables before initialising them.

# Exercises

5 minutes

- Write the `Turtle` class and test it.

We will only start this in class … students will complete this as homework.

# Example 2

- Design a class for a bank account.

| BankAccount |
| --- |
| account number<br>account name<br>balance |
| deposit<br>withdraw |

Some of these slides may be left as practice and self-study.

# UML to Java code

attributes

account number
account name
balance

instance variables

```
int accNo
String accName
double balance
```

operations

deposit
withdraw

methods

```
deposit(double amount)
withdraw(double amount)
```

In Java, instance variables are used to define an object's states (or attributes) and methods are used to define its behaviour.

# A Java class: general template

```
class ClassName {
    // instance variables
    // constructors
    // accessors (or getters)
    // mutators (or setters)
    // service methods
}
```

# Step 1. Instance Variables

```
public class BankAccount {
  private int accNo;
  private String accName;
  private double balance;

  // other code to add ...



}
```

Using **private** for information hiding.

# Step 2. Constructors

```
public class BankAccount{
  private int accNo;
  private String accName;
  private double balance;
  public BankAccount(int accNo, String accName){
      this.accNo = accNo;
      this.accName = accName;
      this.balance = 0.0;
  }
  public BankAccount(String accName, int accNo){
      this.accNo = accNo;
      this.accName = accName;
      this.balance = 0.0;
  }
  // other code to add ...
}
```

Constructor has the same name as the class. User-defined constructor assigns values from arguments.

They are different constructors.

# Step 3. Accessors (getters) and Step 4. Mutators (setters)

```
public class BankAccount {
   // instance variables
   // constructors

   public int getAccountNo() { return accNo; }

   public String getAccountName() { return accName; }

   public double getBalance() { return balance; }

   // other code to add ...
}
```

Provide them only if you allow others to retrieve the states.

```
   public class BankAccount {
      // instance variables
      // constructors
      // accessors (getters)

      public void setAccountName(String accName) {
         this.accName = accName;
      }

      // other code to add ...
   }
```

The account number cannot be set. Directly setting the balance is not allowed; balance changes through **deposit()** and **withdraw()**.

# Step 5. Service methods

- Service methods are used to interact with the data in the object and to change the state of the object.
- In the **BankAccount** example, we can change the state of the balance by making a deposit or a withdraw.
  - In this case, the amount will be passed.

```
public class BankAccount {
    // instance variables
    // constructors
    // accessors (getters)
    // mutators (setters)

    public void deposit(double amount) {
      balance = balance + amount;
    }

    public void withdraw(double amount) {
      balance = balance - amount;
    }

    // other code to add ...
}
```

# Step 6. `toString()` method (1/3)

- To print primitive data (e.g. **`int`**, **`double`**, **`char`**) and **`String`** we use:

```
System.out.println(variableName);
```

- Is it possible to print out an object?
  - What happens if we try to print an object like this?

```
BankAccount myAccount = new BankAccount(11111111,
                                        "John");
System.out.println(myAccount);
```

**BankAccount@119c082**

This looks a bit like an email address! What is it?

- There is an instance of the class **`BankAccount`**, along with an object reference.
- The compiler knows where the object is and what is stored in it, but cannot print it out correctly.
- Because we have not told the compiler how to represent it!

# Step 6. `toString()` method (2/3)

- Methods like **`println()`** or **`print()`** want a **`String`** representation of the object to print.
- To represent the object as a **`String`**, we need to implement the **`toString()`** method.
  - We will actually override the **`toString()`** method defined in the **`Object`** class.
  - This method returns a **`String`** representation of the object.

**`toString()`** method for our **`BankAccount`** class

```
public String toString() {
    return "Account number: " + accNo + "\n"
           + "Account name: " + accName + "\n"
           + "Balance: " + balance ;
}
```

This method must be named **`toString()`** and it must return a **`String`** type.

# Step 6. `toString()` method (3/3)

```
public class BankAccount {
   // instance variables
   // constructors
   // accessors (getters)
   // mutators (setters)
   // deposit method
   // withdraw method
   public String toString() {
    return "Account number: " + accNo + "\n"
            + "Account name: " + accName + "\n"
            + "Balance: " + balance ;
   }
}
```

With the **`toString()`** method, the object can be printed out with a user-defined format.

```
BankAccount myAccount = new BankAccount(11111111, "John");
System.out.println(myAccount);
```

**Account number: 11111111**
**Account name: John**
**Balance: 0.0**

# BankAccount class (in full)

```
public class BankAccount{
  private int accNo;
  private String accName;
  private double balance;

  public BankAccount(int accNo, String accName) {
    this.accNo = accNo;
    this.accName = accName;
    this.balance = 0.0;
  }

  public BankAccount(String accName, int accNo) {
    this.accNo = accNo;
    this.accName = accName;
    this.balance = 0.0;
  }

  public int getAccNo() {
    return accNo;
  }

  public String getAccName() {
    return accName;
  }

  public double getBalance() {
    return balance;
  }

  public void setAccName(String accName) {
    this.accName = accName;
  }

  public void deposit(double amount) {
    balance = balance + amount;
  }

  public void withdraw(double amount) {
    balance = balance - amount;
  }

  public String toString() {
    return "Account number: " + accNo
      + "\n" + "Account name: " + accName
      + "\n" + "Balance: " + balance;
  }
}
```

# Step 7. A test class

```
public class BankAccountTest {
  public static void main(String[] args) {
    BankAccount acc1 = new BankAccount(23142635, “John Smith”);
    System.out.println(acc1);
    acc1.deposit(500);
    acc1.withdraw(100);
    System.out.println(acc1);

    BankAccount acc2 = new BankAccount(“Tom Will”, 38472638);
    System.out.println(acc2);
    acc2.deposit(3000);
    acc2.withdraw(400);
    System.out.println(acc2);
  }
}
```

```
Account number: 23142635
Account name: John Smith
Balance: 0.0
Account number: 23142635
Account name: John Smith
Balance: 400.0
Account number: 38472638
Account name: Tom Will
Balance: 0.0
Account number: 38472638
Account name: Tom Will
Balance: 2600.0
```

# Method Overloading

- Java allows several methods to be defined with the same name, as long as they have different sets of parameters.
- The compiler resolves which particular method is required by examining the signature of the method – its name and the types and sequence of its parameters.
- The return type is NOT used to differentiate methods, so you cannot declare two methods with the same signature even if they have a different return type.
- Examples:

```
public void deposit(double amount, boolean cheque) {
  if (cheque == false) { balance = balance + amount; }
  else {
    // code to be added
  }
}
```

```
public void deposit(double amount) {
  balance = balance + amount;
}
```

# Improving the code …

1. The variable **accNo**: **int** or **String**?
   - Consider the account number **00112612**
2. A better **withdraw** method: do not allow overdraft
3. An even better **withdraw()** method: set up overdraft limit
4. Print some user friendly messages

In our **BankAccount** example …

- A better **withdraw()** method; it does not allow a withdrawal if **amount > balance**
- An even better **withdraw()** method
  - How about setting up an overdraft limit? Try at home …

```
public void withdraw(double amount) {
  if (balance >= amount) {
    balance = balance - amount;
  }
}
```

# Steps when writing classes (1+2)

- Step 1: Think!
  - States and behaviour of the object
  - States → instance variables
    - How many?
    - Type?
    - Private or public?
  - Behaviour → methods
    - How many?
    - Return type?
    - Parameters?

- Step 2: Skeleton (or basic) code
  - Define a class
  - Declare instance variables
  - A set of constructors
    - How many?
    - Parameters?
    - Type?
  - Write a test program (with a `main()` method) to test it
    - Create new objects using provided constructors

# Steps when writing classes (3+4)

- Step 3: Accessors and mutators
  - A set of accessors and mutators
    - How many?
    - Return type?
    - Parameters?
  - Test them in the test program
    - Test each accessor/mutator method

- Step 4: Service methods
  - Write ONE service method first
  - Test it
  - Write another one
  - Test it
  - Etc …
- A method should only do one thing and do it well
  - If a method does too much… consider breaking it down into several smaller methods!

# Steps in writing classes (5+6)

- Step 5: `toString()` method
  - Write it, if necessary
- Now you should have a basic OO program working

- Step 6: Improvement
  - Have a full working basic class first
  - Any improvements?
  - Any better solution?
  - Provide user friendly messages?
  - …