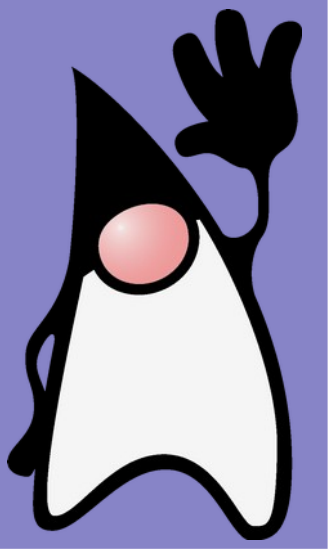
Java

Builder design pattern



Overview

- separating object construction from its representation
- for creation
  - complex objects
  - immutable objects

Example

```
public class BankAccount {
  private String name;
  private String email;
  private String accountNumber;
  private boolean newsletter;

  public static
           class BankAccountBuilder {
  }

  private BankAccount
           (BankAccountBuilder b) {
    this.name = b.name;
    this.accountNumber =
                    b.accountNumber;
    this.email = b.email;
    this.newsletter = b.newsletter;
  }
}
```

```
public static class BankAccountBuilder {
  private String name;  private String email;
  private String accountNumber;
  private boolean newsletter;

  public BankAccountBuilder
       (String name, String accountNumber) {
    this.name = name;
    this.accountNumber = accountNumber;
  }

  public BankAccountBuilder
                  withEmail(String email) {
    this.email = email;  return this;
  }

  public BankAccountBuilder
       wantNewsletter(boolean newsletter) {
    this.newsletter = newsletter;  return this;
  }

  public BankAccount build() {
    return new BankAccount(this);
  }
}
```

Example: usage

```
BankAccount newAccount = new BankAccount
    .BankAccountBuilder("Jon", "22738022275")
    .withEmail("jon@example.com")
    .wantNewsletter(true)
    .build();
```

Advantages/disadvantages

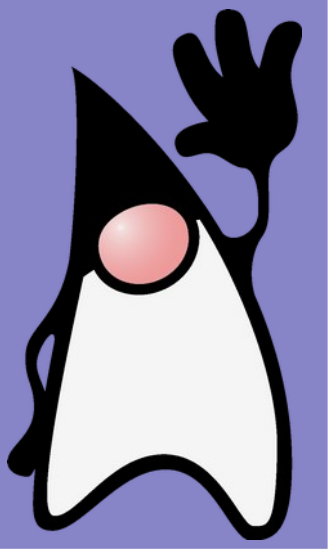
- for complex objects
  - a chain of method calls on the builder class
  - no need for constructors with many arguments
- creating immutable objects
  - no set methods

- disadvantage – to create additional code

- in the std library, used e.g. for StringBuilder and StringBuffer

Java

Prototype design pattern



Overview

- creating new instances via “copying” a *prototype* instance
- when to use
  - many similar objects
  - many copies of a “complex” object

Example

```
public abstract class Tree {
  public abstract Tree copy();
}
```

```
public class PineTree extends Tree {
  @Override
  public Tree copy() {
    PineTree pineTreeClone = new
       PineTree(this.getMass(), this.getHeight());
    pineTreeClone.setPosition(this.getPosition());
    return pineTreeClone;
  }
}
```

```
// usage
PineTree pineTree = new PineTree(mass, height);
plasticTree.setPosition(position);
PIneTree anotherPineTree = (PineTree) pineTree.copy();
anotherPineTree.setPosition(otherPosition);
```

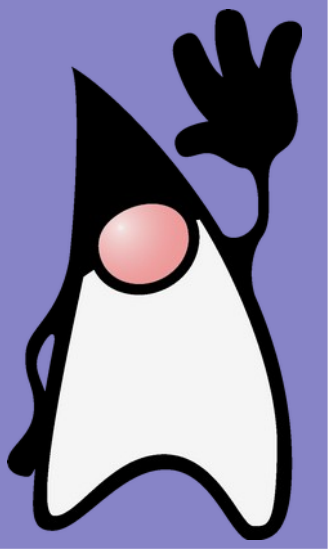
Cloneable and prototype pattern

- Cloneable can be used
- warning – Cloneable by itself does nothing
  - and clone() (as defined on Object) is protected
  - i.e., needs to be overridden as public

- Cloneable and clone() are not ideally designed
  - see the book J. Bloch: Effective Java

Java

What next...



What next

- **NPRG021 Advanced Java Programming**
  - summer 2/2
  - synopsis
  - GUI (Swing, JavaFX)
  - Modules, Reflection API, Classloaders, Security
  - Generics, annotations
  - RMI
  - JavaBeans
  - Java Enterprise Edition: EJB, Servlets, Java Server Pages, Spring,...
  - Java Micro Edition
  - RTSJ, Java APIs for XML, JDBC, JMX,…
  - Kotlin and other “Java” languages
  - Android